PAGODE : a back-end generator for RISC machines

P. Canalda, Lucile Cognard, M. Mazaud, A. Despland

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

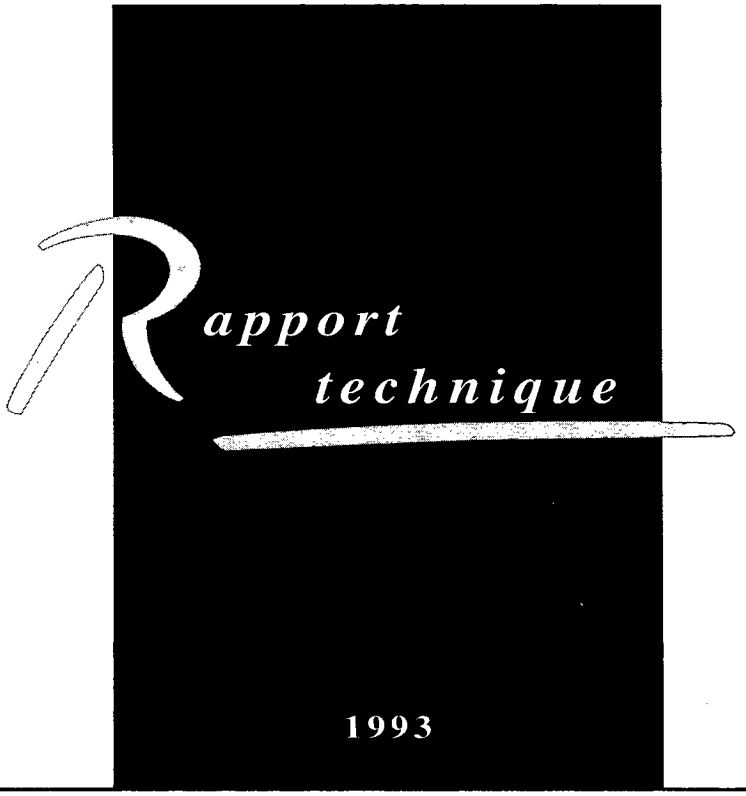
PAGODE : a back-end generator for RISC machines

Philippe CANALDA - Lucile COGNARD
Monique MAZAUD - Annie DESPLAND

N° 152
Mai 1993

PROGRAMME 2

Calcul Symbolique,
Programmation
et Génie logiciel



PAGODE : a back-end generator for RISC machines [1]

PAGODE : un constructeur de générateurs de code pour machines RISC

Philippe CANALDA, Lucile COGNARD and Monique Mazaud [2]
INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex
FRANCE

Annie Despland [3]
LIFO Université d'Orléans
BP 6759, 45067 Orleans CEDEX2
FRANCE

[1]This work was partially supported by ESPRIT Project COMPARE
[2]e-mail:Monique.Mazaud@inria.fr
[3]e-mail:ade@chambord.univ-orleans.fr

Résumé

PAGODE est un constructeur de générateurs de code qui produit les différents moteurs (sélecteur d'instructions, module d'ordonnancement de code et allocateur de registres) d'un générateur de code de facon modulaire. Ce rapport technique présente la première version de ce système et constitue un manuel d'utilisation.

Mots-Clés : génération de code, sélection d'instructions, llocation de registres, ordonnancement de code

Abstract

PAGODE is a back-end generator that produces automatically the various engines of a code generator (instruction selector, scheduler and register allocator) from a target machine specification. This technical report presents the system in its first release and makes up a user's guide.

Keywords: code generation, instruction selection, register allocation, pipeline, code scheduling

Contents

0.1 Introduction 7

1 The SCALA language **9**
1.1 Introduction 9
1.2 Storage bases 10
1.2.1 Error messages 11
1.3 Storage classes 11
1.3.1 Storage class declarations 11
1.3.2 Storage class referencing the memory 12
1.3.3 Error messages 12
1.4 Label classes 12
1.5 Value classes 13
1.6 Access modes 13
1.6.1 Introduction 13
1.6.2 Access mode declarations 14
1.6.3 Error messages 16
1.7 Access classes 17
1.7.1 Access class declarations 17
1.7.2 Error messages 17
1.8 Instructions 17
1.8.1 Instruction declarations 17
1.8.2 Convention 20
1.8.3 Error messages 20
1.9 Directives 20
1.10 The interface specification 21

2 Instruction selection **23**
2.1 Basic concepts 23
2.2 Input and output of the instruction selector engine 24
2.2.1 Error messages 25
2.2.2 MC68000 example 25
2.2.3 SPARC example 26

3 Binding **29**
3.1 Basic concepts 29
3.2 Input and output of the binder engine 29
3.2.1 Input term 29

3.2.2 Output tables . . . 30
3.2.2.1 Binding constraints . . . 30
3.2.2.2 Usedef tables . . . 31

4 The register allocator **32**
4.1 Temporary coloring and spill code production . . . 32
4.1.1 Basic concepts . . . 32
4.1.2 The target machine specification . . . 33
4.1.2.1 Register declarations . . . 33
4.1.2.2 Spill code strategies and target machine specification . . . 34
4.1.2.3 Spill in stack . . . 35
4.2 Universal store rewriting . . . 36
4.2.1 Basic concepts . . . 36
4.2.2 Target machine specification . . . 36
4.2.3 The convert constructor . . . 36
4.2.4 Error messages . . . 37

5 The scheduler **39**
5.1 Basic concepts . . . 39
5.1.1 Pipeline technique . . . 39
5.1.2 Short overview of the PAGODE scheduler . . . 39
5.2 The target machine specification . . . 40
5.2.1 Specifications related to data hazards . . . 40
5.2.2 Specifications related to structural hazards . . . 42
5.2.3 SPARC declaration examples . . . 44
5.2.4 Specifications related to control hazards . . . 45
5.3 Input and output of the scheduler engine . . . 46

6 Overall structure of the code generators produced **48**
6.1 Code generators for CISC machines . . . 48
6.2 Code generators for RISC machines . . . 48
6.2.1 Integration of register allocation and scheduling . . . 48
6.2.2 Specification for integration . . . 50
6.2.3 Error messages . . . 52
6.3 Input and output of the prepare engine . . . 52
6.3.1 Input of the prepare engine . . . 52
6.3.2 Output of the prepare engine . . . 53

7 Creating a code generator **54**
7.1 Installation of the PAGODE system . . . 54
7.2 Creating a code generator . . . 54
7.3 Code generation . . . 58

Bibliography **60**

ANNEXES **61**

A Derivation chains for a SPARC PMIR **62**

List of Figures

0.1 The PAGODE System. 8

6.1 The overall structure of a CISC back-end. 49
6.2 The overall structure of a RISC back-end. 51

0.1 Introduction

PAGODE is a back-end generator based on tree rewritings that takes as input a target machine description using a special purpose language called SCALA and produces a code generator that performs instruction selection, register allocation and scheduling and produces assembly code.

A target machine specification written in the SCALA language must be hierarchically organized into concepts corresponding to the main features of the instruction set processor : locations, addressing modes and instructions.

The semantics of each concept of the target machine specification is given by a template which is a term of an ADT (abstract data type). Such an ADT is specific to a given target machine and must be specified by the compiler writer while writing the SCALA specification. An abstract data type is a collection of sorts with their axioms. A sort is similar to a type with its operators. A term is an object of a given type built from these operators. The SCALA specification of a target machine requires only the syntactic part of the ADT. The axioms can be used by the compiler writer to specify some special cases (see section 1.6.2).

The terms built on this ADT are called PMIR (PAGODE medium intermediate representation), they are used as input to a back-end produced by the PAGODE system. In the PMIR input term to the back-end, instructions act on cells via operators denoting access paths to cells.

For CISC machines, the code generation process is divided into three steps : instruction selection, binding and register allocation. Additional steps of scheduling are provided for RISC machines. The engines that perform those steps are automatically produced by the PAGODE system.

The instruction selection process applies a set of rewriting rules driven by tree templates derived from the target machine specification to the PMIR term. Basically each instruction of the PMIR is matched with an instruction template. In the context of such a template, the operands are matched with addressing mode templates. If an operand does not match any addressing mode template and a subterm does, then the location depicted by the subterm is stored in a "temporary resource" using a universal store. The PMIR is rewritten using this temporary resource.

The output of the instruction selector belongs to the LIR (low level intermediate representation) of the target machine. It includes universal assignments, references to temporaries, addressing modes and instructions in their "canonical" representation.

The second code generation step, called binding, sets constraints on those temporaries according to the whole context and performs life-time analysis and packing on temporaries.

A LIR term can be either the input of the register allocator or the scheduler.

The register allocator needs knowledge of the actual resources of the target machine that have been declared in the target machine specification. If all registers have been assigned, it is necessary to spill some of them, thus a spill code strategy must be identified. That is the reason why a "spill code" construction has been added to the SCALA language, in fact it is a compiler strategy that uses both spill instructions and data storage directives.

The scheduler works on the LIR using data dependence analysis on the target machine resources and attributes on the use of functional units that have been added to the target machine specification.

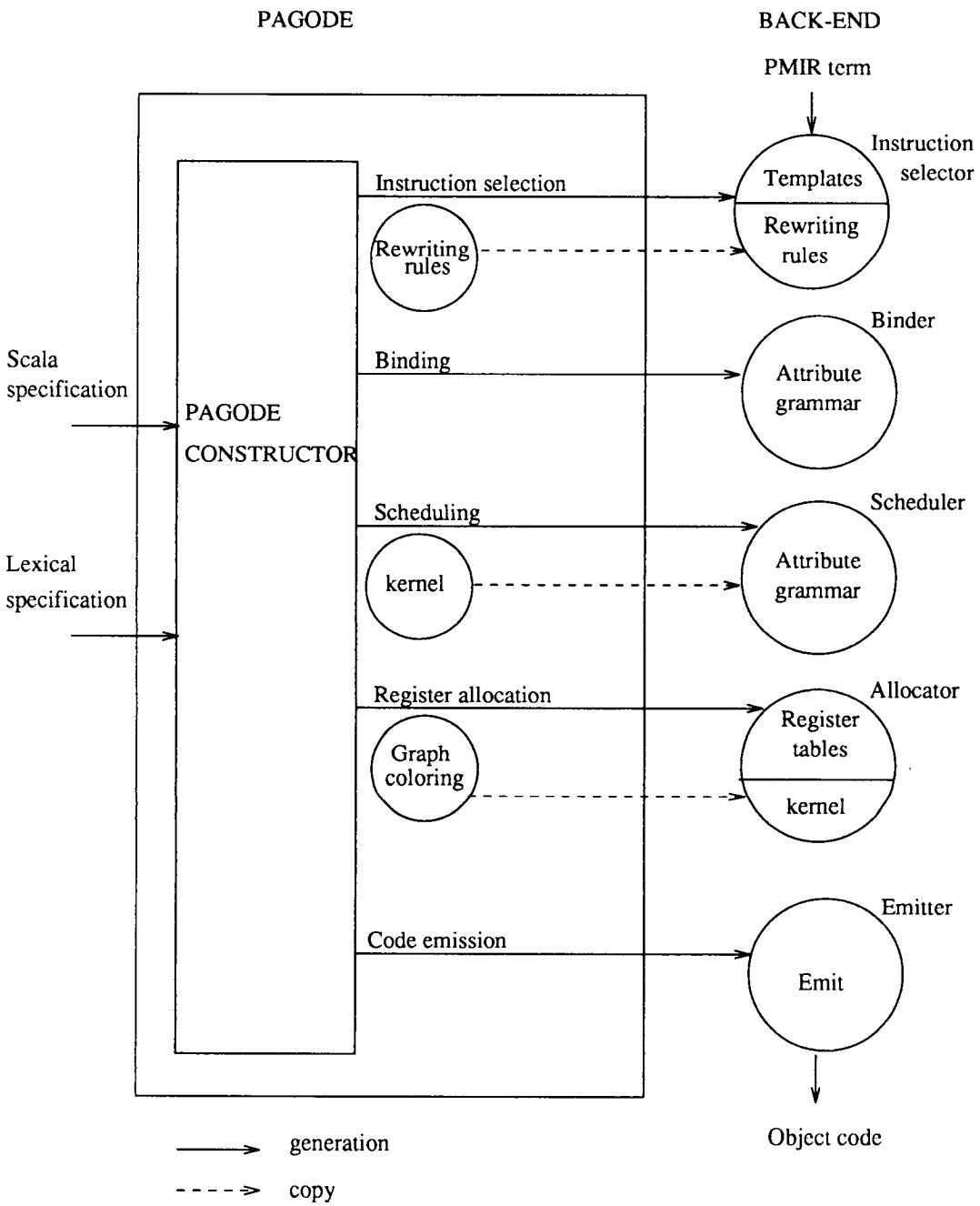


Figure 0.1: The PAGODE System.

Chapter 1

The Scala language

1.1 Introduction

The PAGODE system provides the special purpose language SCALA to specify the instruction set of a target machine. A SCALA specification is handled by the SYNTAX system which provides automatic construction of scanners and parsers for a given language. The definition of SCALA has been done using BNF rules and annotations aiming to give names to the abstract tree built during parsing. The code generator writer is guided during the development of a given target machine specification by the error messages of SYNTAX since this system implements in its parser and scanner a powerful *error recovery* mechanism [BJ 87].

A full syntactic definition of SCALA will be found in the next release of the reference manual. Currently, specifications given as examples of target machine and various sections described in this chapter can be considered as a sufficient introduction to the syntactic form of various sections of this language. In the sequel of this document, all keywords of SCALA are in bold letters.

A target machine description can be easily deduced from the handbook. The basic concepts of an assembly language are described by specific constructs of SCALA : *storage bases*, *storage classes*, *label classes*, *value classes*, *access modes*, *access classes* and *instructions*. A target machine description is hierarchically structured in seven levels :

- *storage bases* are used to declare storage descriptions (sets of available locations, i.e. registers, memory locations) :
- *storage classes* are logical partition of storage bases : there must exist as many storage classes as there are ways to gather storage base elements :
- *label classes* are used to depict operands that are labels ;
- *value classes* are used to depict operands that are immediate constants ;
- *access modes* correspond to addressing modes description (various ways to access locations) ;
- *access classes*: since an operand of an instruction may accept several access modes, the access modes are assembled into access classes :
- *instructions* belong to the instruction set processor.

The semantics of each construct is expressed using a term of the target abstract data type (**ADT**) of the target machine and various attributes that describe the properties of the construct such as format, space cost, time cost, size ....

This chapter describes the various concepts used to depict a CISC machine, the features of the SCALA language related to the specific properties of a RISC machine are optional. They will be described entirely in the fifth chapter.

Since the occurrences of a construct for various sizes of the addressable units are often nearly identical, a solution proposed to deal with large algebraic specifications is the use of parameterization and instantiation mechanisms [DMR 90a]. Such mechanisms fit very well with our machine specification language. The compiler writer can factor out some instances of a given construct in a generic pattern followed by the possible values of the generic parameters of the pattern. The system derives from this declaration as many occurrences of the construct as there are sizes of addressable units asssociated with it. The instantiation mechanism is bound to a name generation mechanism. Throughout the paper the following notations will be used :

- If **n** is a name and **L** is a variable, when **L** is instantiated by **v**, **n!L** builds the name **n_v**.
- <S V> means that **V** is a variable or a constant of sort **S**.

A basic hypothesis of the PAGODE system is that the measure unit for the sizes is the byte.

1.2 Storage bases

A storage base is defined as the set of the smallest addressable units of a physical storage.

For example the CYPRESS SPARC CY7C601/612 architecture has 136 32 bits-wide registers that are divided into a set of 128 window registers and a set of eight global registers. The 128 window registers are grouped into eight sets of 24 registers called windows that overlap on eight registers. At any given time, a program can address 32 active registers: 24 window registers and eight globals denoted by %gx. The window registers are divided into 8 input registers denoted by %ix, 8 local registers denoted by %lx and 8 ouput registers denoted by %ox, their names are listed in the **Symbolic_notation** declaration. A list of register names is allowed as symbolic notation. Each item of the list can be depicted by a string followed by an integer interval. An item of this list may include the declaration of an alias for a register name, it is introduced by the keyword **alias**. Thus in the declaration below :

- %fp is an alias of %i6, among the input registers called %i1, %i2, %i3 ...
- %sp is an alias of %o6, among the output registers called %o1, %o2, %o3 ...

Thus, the compiler writer must declare the following storage base :

```
Storage_base GPR          - - General purpose registers
    Symbolic_notation
        'REG' is %g(0..7)
        ,%i(0..7) with %fp alias %i6
        ,%o(0..7) with %sp alias %o6
        ,%l(0..7)
    $ base_units    =    4
    $ free          =    %l(0..7)
End
```

The registers available for register allocation are specified into the **$free** attribute, that is the case of the local registers only. For single precision, floating-point operands are placed in floating point registers. 32 floating-point registers are available, they are called %f1, %f2, %f3, %f4 ....

```
Storage_base FREG        - - Floating point registers
    Symbolic_notation
        'FREG' is %f(0..31)
    $ base_units    =    4
    $ free          =    %f(0..31)
End
```

The PAGODE system assumes that there exists on every target machine an homogeneous and uniform storage base corresponding to the memory. By convention, this storage base is called **MEM**. It is not necessary to declare it in a SCALA specification.

1.2.1 Error messages

The PAGODE constructor checks that the second element of an alias declaration belongs to the previous enumeration of resources. For instance, if the GPR storage base includes the following specification part

- ```
  Storage_base GPR          - - General purpose registers
      Symbolic_notation
          'REG' is %g(0..7)
          ,%i(0..7) with %i6 alias %fp
          ...
          ...
      End
  ```

  then the PAGODE constructor would emit the following error message:

- Error : In the context of the storage base GPR, the alias name %fp must belong to the previous enumeration..

1.3 Storage classes

1.3.1 Storage class declarations

For a given storage base, the compiler writer must describe as many storage classes as there are ways to gather storage base elements to represent logical storage units. Thus, we define a new concept: the storage class based on a storage base. For instance, an access to a register may designate a byte operand, a word operand or a longword operand.

A storage class occurrence is characterized by the following properties:

- its denotation introduced by the keyword **Denotation.**
- its storage base viewed as an attribute of a storage class is introduced by the keyword **$Base.**
- basic operations introduced by the keyword **Operations.** The dereference operation, i.e. the access operation to the contents of an element of the storage class begins with the keywords **dereference is.** The cell constructor operation begins with the keyword **cell_constructor is.**
  ```

When there exists an alternative on the sort of the contents of a storage class element, it is introduced by the keyword **Union**. In the following example, the contents of a general purpose register can be either an address or a 13 bit value, or a 22 bit value.

In the SCALA language, the storage class construct is described using a predefined keyword for each of these properties. Let us consider the storage classes related to the general purpose register storage base. As an access to a general register for the SPARC may represent an access to a word operand or a double word operand, the compiler writer must declare two storage classes respectively : the gpr_W and gpr_D storage classes. The compiler writer can declare a generic pattern of a *general purpose register* storage class using generic names. The **Instances** part of the declaration includes the information needed by the name generation mechanism to build the actual names.

```
Storage_class gpr
    Denotation < gpr!size reg>
    Attributes
        $Base = GPR
    Operations
        dereference is
            cont_of_gpr!size      : gpr → Union(address, value_13 , value_22)
        cell_constructor is
            designates_gpr!size   : gpr → gpr
    Instances
        size in { W, D}
            case size is
                W                 : size = 4 base_units
                D                 : size = 4 double base_units
            End case
End
```

From this pattern, the system deduces the two descriptions of actual storage classes. The size of a given storage class is specified counting the base_units occupied. The keyword **double** means that the PAGODE system will consider a pair of general purpose registers.

1.3.2 Storage class referencing the memory

There must exists a storage class based on the **MEM** storage base. The information related to this storage class are used to generate the scheduler.

1.3.3 Error messages

- Warning : Unknown storage base ...in the declaration of the storage class....
- Error : Inconsistency between the number of instances and the number of case choices in the declaration of the storage class ....

1.4 Label classes

Some labels are used in addressing modes or in instructions, they must be specified, too. They are introduced by the keyword **Label_class**. In the PAGODE system, the hypothesis that all labels belong to a set of labels called **UNIV_LABEL** holds. **UNIV_LABEL** is a keword of the PAGODE system.

Label_class data_label
Denotation <data_label lab>
Attributes
$Base = UNIV_LABEL
Operations
const_dlabel : data_label — int_immediate_value
add_lab : address, int_immediate_value — address
Symbolic_notation
lab **in** DATA_LABEL
End

1.5 Value classes

The assembly language of the target machine uses immediate values as operands of instructions, as offsets or as displacements of addressing modes. According to the target machines these values take different sizes.

Thus on the SPARC architecture. 13-bit values are used to specify a frame pointer with displacement addressing mode. On the MC68000, the similar addressing mode takes a 16-bit value as offset.

A similar comparison holds if one considers the relative offsets to the program counter for branches:

1. on a MIPS, branches have 16-bit offsets relative to the program counter.
2. on a SPARC, branches have 22-bit offsets relative to the program counter.

That is the reason why each different value size must be declared as a different sort with its operators, this concept is called a value class. A sort for an immediate value is introduced by the keyword **Value_class**. The same hypothesis as for labels holds, i.e. there exists a general set of integers called **UNIV_INTEGER**.

The specification of a 13-bit value for the SPARC is as follows:

Value_class value_13
Denotation <value_13 val>
Attributes
$Base = UNIV_INTEGER
Operations
const_value_13: value_13 — value_13
add_13 : value_13 . value_13 — value_13
sub_13 : value_13 . value_13 — value_13
mul_13 : value_13 . value_13 — value_13
Symbolic_notation
val **in** INTEGER_13
End

1.6 Access modes

1.6.1 Introduction

Let us consider an assignment statement of A to B, we shall state in the sequel that A is the source operand(*r*-value) and B the destination operand (*l*-value) of the assignment. In an instruction,

an operand is an element of a set of access modes. Whereas an access mode in source position designates the contents of a location, it designates the location itself in destination position.

An access mode can be defined as an addressing mode of a virtual machine on which all standard addressing modes are available, i.e. direct addressing mode, indexed addressing mode, indirect addressing mode[1].

1.6.2 Access mode declarations

A particular machine has several addressing modes. For a given addressing mode of the machine, the compiler writer must define as many access modes as there are associated storage classes. An access mode is specified by :

- *a canonical form representating the access mode, including its name and its parameters. These parameters are formal storage or value classes. It is introduced by the keyword* **Canonical_form**.
- *its related attributes : length, format, space cost and time cost. They are respectively introduced by the keywords:* **$length**, **$fmt**, **$space_cost**, **$time_cost**.
- *a template that describes the access path to the corresponding operand : the operand in source (resp. destination) position is defined by the term obtained by applying the dereference (resp. the cell constructor) operation to this template. It is introduced by the keyword* **Template**.

The *canonical form* of an access mode has parameters that characterize a given addressing mode. For instance, let us consider an indirect with displacement addressing mode on a SPARC, it takes as operands a general purpose register and a 13 bit offset. Now consider the similar addressing mode on a MC68000, it requires an address register and a 16 bit offset.

On the SPARC architecture, the related access mode when the size of the operand is half word can be specified as follows :

```
Access_mode
    Canonical_form                  - - Indirect with displacement access mode
        disp_am_H ( <gpr_W reg> , <value_13 val>)
    Attributes
        $length     =   H               - - length of the addressable unit
        $fmt        =   ~ reg + val ~   - - Assembly language format
        $space_cost =   0
        $time_cost  =   1
        Template
            index ( cont_of_gpr_W (<gpr reg>)
                    , const_value_13 (<value_13 val>))
End
```

The template introduced by the keyword **Template** has no meaning in itself. The PAGODE constructor must handle that feature to build two templates from this definition : a template in *r*-value (i.e. in source position), a template in *l*-value (i.e. in destination position). For the previous disp_am access mode definition, the PAGODE constructor works as follows. From the signature of the index

[1] Notice that such an addressing mode does not always exist on a real target machine, that is the case of the SPARC for which there exists no indirect addressing mode. Since such a construct is necessary in the PMIR, we shall give a solution to deal with this issue in the sequel.

operator, the constructor returns an address whose dereference operator is "cont_of_address" and whose cell constructor is "designates_address", the system derives the two following access modes, respectively in destination and source position :

```
designates_address_H (
    index (cont_of_gpr_W (<gpr reg>), const_value_13 (<value_13 val>)))
cont_of_address_H (
    index (cont_of_gpr_W (<gpr reg>), const_value_13 (<value_13 val>)))
```

In some cases, an access mode exists only in a given position, either source or destination position. The SCALA language provides two specific constructs to handle those cases. When the keyword **Template** is followed by the keyword **Src**, it means that the access mode exists only as an *r*-value. Thus an immediate value, on the MC68000, can be used only as an *r*-value, i.e. in source position.

```
Access_mode
    Canonical_form
      immediate_val_am!size (<immediate_value!size val>)
    Attributes
      $length       =   size
      $fmt          =   ~val~
      $space_cost   =   1
      $time_cost    =   case size is
                            B , W : 4
                            L : 8
                        End case
    Template
      Src           =   const_value_!size ( <immediate_value val>)

    Instances size in {B, W, L}
End
```

When the keyword **Template** is followed by the keyword **Dst**, it means that the access mode exist only as a *l*-value. That is the case of the addressing modes used in branches. In the MC68000 specification, the specification of a destination of a branch can be specified by the following access mode :

```
Access_mode
    Canonical_form
      relative_clab_am (<code_label lab>)
    Attributes
      $length       =   L
      $fmt          =   ~lab~
      $space_cost   =   1
      $time_cost    =   8
    Template
      Dst           =   const_clabel_W (<code_label lab>)
End
```

As for the storage class construct, the compiler writer can define a formal access mode. Among the numerous addressing modes of the MC68000, let us consider *the indirect with displacement*

addressing mode. This access mode has instances introduced by the keyword which depend on the size of the location indirectly accessed in source position. Thus the compiler writer defines a generic access mode pattern "disp_am!size" parameterized by the size. This leads to three templates in source position (respectively in destination position) when the size is instantiated by {B, W, L}. The *indirect with index* access mode has instances which depend on the size of the location indirectly accessed in source position.

```
Access_mode
    Canonical_form        - - Indirect with displacement access modes
        disp_am!size (<aregister_L reg> , <immediate_value_W val>)
        when val = 0
                disp_am!size (<aregister_L reg>, <immediate_value_W 0>)
    Attributes
        $length =   size                - - length of the addressable unit
        $fmt    =   ~ val (reg) ~       - - Assembly language format
                    when val = 0 ~(reg)~
        $space_cost = 1
        $time_cost = case size is
                    B . W : 8
                    L : 12
                    End case
    Template
        index ( cont_of_areg_L (<aregister reg>)
                , const_value_W (<immediate_value val>))
        when val = 0
                cont_of_areg_L (<aregister reg>)
    Instances
        size in {B, W, L}
End
```

A special purpose construct called **when** has been introduced in SCALA to handle PMIR terms that cannot be recognized by any access mode template. Let us consider an indirect operation of the PMIR for a SPARC, it looks like the following:

cont_of_address (cont_of_areg_L (<aregister reg>)) in source position.

Similarly, an indirect operation in destination position corresponds to the template:

designates_address (cont_of_areg_L (<aregister reg>))

These PMIR terms can be recognized by the disp_am access mode when val equals zero owing to the following axiom on the index operator:

index (address, value'0') = address

1.6.3 Error messages

The PAGODE constructor checks whether a storage class occurring in an access mode declaration has been previously declared. When this condition does not hold the following error message is emitted:

- Error : Unknown storage class ...in the declaration of the access mode ....

1.7 Access classes

1.7.1 Access class declarations

The operands of an instruction are access classes which are defined as sets of access modes. An access class is introduced by the keword **Access_class**. It can be also specified by a generic pattern including the instantiation of its elements. there are as many instances of a generic access class as there are possible sizes of operands.

Access_class
<Reg_or_imm_access AM >
= gpr_am_W (< gpr_W reg >)
= int_freg_am_W (< int_freg_W reg >)
= value_13_am (< value_13 val >)
= ...
End

The instances of a generic access class are also introduced by the keyword **Instances**.

Access_class
<Gpr_access!size AM>
= gpr_am!size (<gpr!size reg>)
= int_freg_am!size (<int_freg!size reg>)
Instances
size **in** { W, D}
End

1.7.2 Error messages

The PAGODE constructor checks whether an access mode occurring in an access class declaration has been previously declared. When this condition does not hold the following error message is emitted :

- Error : Unknown access mode ...in the declaration of the access class ....

1.8 Instructions

1.8.1 Instruction declarations

An instruction may be characterized by the following properties :

- *a canonical form representation of the instruction including the access classes to which the instructions apply. It is introduced by the keyword* **Canonical_form**.
- *its related attributes : length, format, modification and time cost.*
- *the template describing what is performed by the instruction (it is a term of the abstract data type). It is introduced by the keyword* **Template**.

The format attribute introduced by the keyword **$format** corresponds to the syntax in the assembly language. The time cost is expressed into a number of clock cycles of the target machine, this specification begins with the keyword **$time_cost**. The space cost of an instruction is supposed to be one word. It is a hypothesis of the PAGODE system.

Let us consider the SPARC instruction "store" which stores the contents of a general purpose register in a memory location. We obtain the following specification :

```
Instruction
    Canonical_form
        st (<Gpr_access_W AM1>, <All_access_W AM2>)
    Attributes
        $length        =  W
        $fmt           =  ~st  $fmt(<Gpr_access_W AM1>)
                               , $fmt(<All_access_W AM2>) ~
        $modification  =  store
        $time_cost     =  5
        Template
            assign_W (src (<Gpr_access_W AM1>), dst (<All_access_W AM2>))
    End
```

The **Attributes** keyword introduces various properties of the instruction. The **$length** attribute of an instruction must be the length of the operand in destination position. This length is generally the same that the size of its operands. Nevertheless there exists some cases in which the result has not the same size; that is the case of a multiply instruction on the MC68000 where the result is located on a double word :

```
Instruction
    Canonical_form
        muls (<Data_access_W AM1>, <Dregister_access_L AM2>)
    Attributes
        $length        =  L
        $fmt           =  ~MULS $fmt(<Data_access_W AM1>)
                                  , $fmt(<Dregister_access_W AM2>) ~
        $modification  =  binary_arith
        $time_cost     =  70
        Template
            assign_L (mul (
                           src (<Dregister_access_W AM1>)
                           , src (<Data_access_W AM1>))
                      , dst (<Dregister_access_L AM2>))
End
```

The **$fmt** is a string included between ~ which uses the formats of the parameters of the instruction. It is used to emit the assembly code.

Instructions have been split into various families by means of the attribute **$modification**. This allows one to build the control flow graph needed by the register allocator and the scheduler. The notion of family is not only related to the control flow graph, it is used by the instruction selector as well as by the register allocator and the scheduler to perform their specific jobs. Since the families of interest are not exactly the same for all of them, a rather fine grain of families has been defined.

- assignment families :

  required by both instruction selector and scheduler.

  - **spill**: a load or a store for CISC machines ;
  - **load**: assignment from memory to register must be specified for RISC machines ;

- **store**: assignment from register to memory must be specified for RISC machines ;[2]
- **binary_arith**: assignment for arithmetic computations where general purpose resources are involved ;
- **unary_arith**: assignment for arithmetic computations where general purpose resources are involved ;
- **convert**: for conversion operations ;
- **special_arith**: assignment for arithmetic computations where specific resources like SP or PC are involved ;

- branch families :

  to compute the control flow of the LIR output of the instruction selection, and of the LIR input of the scheduler ;

  - **branch**: for inconditional branches ;
  - **cond_branch**: for conditional branches ;
  - **proc_call**: for procedure calls ;
  - **return_proc**: for return from procedure call ;

- miscellaneous :

  - **test**: for all kinds of comparisons
  - **put_label**: label setting is important to build the control flow graph ;
  - **directive**: for assembly directives ;
  - **noop**: for no-operation instructions.

The template expresses the semantics of this instruction : the source operand of the instruction (i.e the content of the general purpose register) is stored in the destination of the instruction (i.e a memory location designated by the access class All_access_W).

Nearly every instruction of the target machine may be applied to the different lengths of its operands. In order to avoid the repetition of such descriptions, the compiler writer specifies a pattern of an instruction and its instances. Let us consider the *addi* instruction on the MC68000 which corresponds to an immediate addition operation. The size of the instruction may be specified to be a byte, a word or a longword. The instances of a generic instruction are also introduced by the keyword **Instances**.

Instruction
  Canonical_form
    addi!size (<Immediate_access!size AM1>, <Altdata_access!size AM2>)
  Attributes
    $length = size
    $fmt = ~ ADDI.**$length** **$fmt** (<Immediate_access!size AM1>)
      , **$fmt** (<Altdata_access!size AM2>) ~
    $modification = binary_arith
    $time_cost = **case** size **is**
      B , W : 4

[2] the load or store attribute is used by the scheduler for RISC machines.

```
                    L : 8
                End case
Template
        assign!size (
            add!size (src ( <Altdata_access!size AM2>)
                    , src ( <Immediate_access!size AM1>))
        , dst (<Altdata_access!size AM2>))

Instances size in {B, W, L}
End
```

The time cost may depend on the size of the instruction, a **case** construct is available in the language for that purpose.

1.8.2 Convention

- The source parameters of the canonical form of an instruction must always be declared before the destination parameter. This hypothesis is not used by the instruction selector but by the register allocator (see section 4.2.2) and the prepare engine (see section 6.1).
- Several instructions can have the same name. In that case, the PAGODE constructor renames these names from the second occurrence of an instruction. The second occurrence of a given name is renamed adding "1" to the initial name, the third occurrence is renamed adding "2" ....

  This convention must be known by the compiler writer if the PMIR includes instructions in their canonical form.

1.8.3 Error messages

- The PAGODE constructor checks whether an access class occurring in an instruction declaration has been previously declared. When this condition does not hold the following error message is emitted :

  Error : Unknown access class ...in the declaration of the instruction ....
- The PAGODE constructor checks whether a storage class occurring in an instruction declaration has been previously declared. When this condition does not hold the following error message is emitted :

  Error : Unknown storage class ...in the declaration of the instruction ....

1.9 Directives

Most assemblers provide "assembler directives" which are actually instructions for the assembler rather than the processor. Since they must be produced by the compiler too, the target machine specification must include a specification of such directives. Their syntactic form begins with the keyword **Directive** and includes only a **Canonical_form**, a length and a format.

The EQU directive of the MC68000 is used to equate a number to a symbol. The following specification holds when this number is specified by an immediate value :

```
Directive
    Canonical_form
        equ ( immediate_dlab_am_W (<data_label_W lab>)
            , immediate_val_am_W (<immediate_value_W val> ))
    Attributes
        $length =   W
        $fmt    =   ~ lab EQU val ~
End
```

The data storage directive "DS" of the MC68000 is used to reserve space in memory. Since the length specification determines whether bytes, words or longwords are reserved, this directive is specified as follows:

```
Directive
    Canonical_form
        data_storage!size ( immediate_val_am_W (<immediate_value_W val>))
    Attributes
        $length =   size
        $fmt    =   ~ DS.$length ~
    Instances
        size in {B, W, L}
End
```

1.10 The interface specification

Basically, the instruction selector matches each instruction of the PMIR with an instruction template. In the context of such a template, the operands are matched with addressing mode templates. If an operand does not match any addressing mode template and a subterm does, then the location depicted by the subterm is stored in a "universal resource" using a universal store instruction. The PMIR is rewritten using this universal resource [DMR 90a].

A universal resource stands for any actual resource of any sort associated to registers. The built-in sort **temporary** denotes the sort related to universal resources. A universal store stores the contents of a cell in a resource of sort **temporary**, this operator is denoted by **Univ_assign**. The addressing mode for a temporary : **temporary_am** is also provided by the PAGODE system.

A **Univ_assign** is a generic assignment operation. It means that the content of a location is stored in a universal temporary resource. One can say that it is a generic assignment in the sense that this operator can be overloaded by different operators of the PMIR.

The system needs to associate the internal names such as **temporary_am**, **Univ_assign** with the actual names of the machine specification that become possible synonyms during the 2rewriting process. For that purpose, the compiler writer uses an interface declaration construct introduced by the **Interface declaration**. For the MC68000 it follows:

```
Interface declaration
  $Bind =   {aregister, dregister}

  Access_class
        <Temporary_access AM>
            = dreg_am!size (<dregister!size temp>)
            = areg_am!size (<aregister!size temp>)
                    where size in {W, L}
```

```
        Instances
            size in {B, W, L}
    End

    Univ_assign_equivalence
        Univ_assign : assign!size where size in {B, W, L}
            $space_cost = 0, $time_cost = 4
End
```

The PAGODE system needs to know the proper storage synonyms of a temporary. This information can be deduced from the declaration of the **$Bind** attribute.

PAGODE also needs to know the access modes that are correct aliases for the temporary access mode, these informations are deduced from the **Access_class** construct in the interface specification part.

The **Univ_assign_equivalence** introduces an alias between the Univ_assign operator and the assignment operator of the PMIR (with its right sizes). Costs must also be associated to this operator, this costs can be computed as the average of the costs of spill instructions.

Chapter 2

Instruction selection

In order to achieve the instruction selection task, the PAGODE constructor feeds the selector with several sets of templates that are automatically derived from the target machine specification. The instruction selector includes two parts: a bottom-up tree pattern matching [HO 82] and a reducer [RR 92]. These two engines come from the theory of rewriting terms with automata using bottom-up tree pattern matching.

2.1 Basic concepts

The goal of the instruction selection step is to identify each modification of the PMIR with the canonical form of the instruction of the target machine. Such operation is also called "normalization problem" in the classical rewriting terms theory. It runs in two steps:

- the first one identifies a set of rewriting rules that can be applied to a PMIR modification. A composition of rewriting rules used to get such a PMIR modification will be called derivation chain in the sequel.
- the second one selects the cheapest derivation chain using the cost attributes in the target machine specification.

The bottom-up matching process of the PMIR is carried out until each instruction of the PMIR may be reduced into an instruction canonical form: we call LIR (low intermediate representation) the terms that are basically a sequence of instruction canonical forms.

For each instruction, in the context of an instruction template (respectively a canonical form of an instruction), operand subterms are matched with access mode templates (respectively with access mode canonical forms). If they are leaves of a canonical form of an instruction, they have to be in their canonical form and the reduction process is unnecessary. Otherwise, if the whole operand subterm matches an access mode template, the tree pattern matcher takes the tree where the operand is replaced by the canonical form of the access mode as a new goal. If it is only an inner subterm that matches an access mode template, the tree pattern matcher takes as a new goal the tree where the inner subterm is reduced to a temporary location.

The space cost of a derivation chain is computed adding the space cost of each instruction of the LIR, using the following definition for the space cost of an instruction of the LIR:

space_cost (instruction) = $\sum$ **$space_cost** (access_modes) + 1

The time cost of a derivation chain is computed adding the time cost of each instruction of the LIR, using the following definition for the time cost of a LIR:

time_cost (instruction) = $\sum$ **$time_cost** (ins) + **$time_cost** (access_modes)

The cost of the derivation chain may be increased by the cost of a universal store and the costs of its operands. The action associated to this reduction is the production of a universal store tree of the location designated by the recognized access mode into the temporary location.

The bottom-up matching process goes on in order to compute all the possible derivation chains that lead to an instruction in canonical form with their respective costs. The system selects the cheapest one. Once a derivation chain is selected, the reductions involved are effectively performed as well as the associated actions.

The modification tree is flattened into a sequence of universal store trees followed by an instance of the canonical form where the operands have been settled conforming to the instruction template that matches.

A universal store has the **Univ_assign** operator as root. It stores the contents of a cell designated by an access mode in source position in a resource of sort **temporary**. A universal store is another kind of instruction of the LIR.

For each PMIR term, the rewriting algorithm needs to know the boundary where the access mode pattern matching must stop and where the instruction pattern matching must begin. On one hand, the rewriting system uses the set of instruction templates where each leaf refers to an access class, i.e a set of access modes to set the context. On the other hand, it uses the set of access mode templates for the reduction process and the set of arithmetic instructions.

A PMIR term representing an expression can contain embedded arithmetic and access path operators. It is not always possible to produce the related arithmetic instruction instead of the universal store assignment. For that purpose, the instruction selector needs to know for which instructions the **$modification** attribute equals to binary_arith, unary_arith, special_arith or to convert.

The instruction selection algorithm applies the set of rules specified in [DMR 90b].

2.2 Input and output of the instruction selector engine

The input of the instruction selector is a PMIR term that may include actual registers, in that case the compiler must be aware that this register must not appear in the list of free registers of the related storage base.

Universal resources may also appear in the PMIR via the temporay_am access mode. Such an occurrence may have been produced as output of an optimizing engine working on the PMIR. The output of the instruction selector is a LIR term, i.e a sequence of instructions in their canonical form where the access classes have been replaced by the right access mode or a temporay_am access mode.

The instruction selector of PAGODE generates temporary resources which do not clash with the actual and temporary resources in the PMIR. The LIR includes different kinds of temporaries:

- those coming from the PMIR. Their life time may be greater than that of a basic block.
- those coming from the instruction selector. Their life time is local to a basic block. Their name begins by the string "&=&_tmp" followed by a sequence of integers. To improve the legibility of this manual &=&_tmp will be replaced by tmp in the sequel.

2.2.1 Error messages

The PAGODE instruction selector checks whether a PMIR modification can be rewritten according to the templates derived from the target machine specification. When this condition does not hold one of the following error messages is emitted :

- Instruction selection failed. The system can not rewrite the PMIR tree of root ....
- Instruction selection failed. Rewriting interface fails because the system cannot find an access mode of root ....
- Instruction selection failed. The rewriting system is waiting for either an interface rule or another access class definition. The system can not rewrite the PMIR tree of root ....

2.2.2 MC68000 example

Let us consider the MC68000 PMIR term that corresponds to the assignment X := X + T where X and T are local to a procedure. The offsets of X and T from the content of the frame pointer are respectively -4 and -8 :

```
exec (seq (
    assign_L (
            add_L (
                cont_of_address_L (
                    index ( cont_of_areg_L ( denotation ( aregister, a6 ))
                        , const_value_W ( denotation ( immediate_value, -4 ))))
                , cont_of_address_L (
                    index ( cont_of_areg_L ( denotation ( aregister, a6 ))
                        , const_value_W ( denotation ( immediate_value, -8 )))))
            , designates_address_L (
                index ( cont_of_areg_L ( denotation ( aregister, a6 ))
                    , const_value_W ( denotation ( immediate_value, -4 )))))
    ))
```

The bottom-up tree pattern matching system may generate several derivation chains. Among the solutions, two derivation chains derived from the previous PMIR term are shown below :

- seq (

  $Univ\_assign^{time\_cost=4}_{space\_cost=1}$ (

  $disp\_am\_L^{time\_cost=12}_{space\_cost=1}$ <aregister_L a6> <immediate_value_W -8>

  , $temporary\_am^{time\_cost=0}_{space\_cost=0}$ <temporary tmp0>)

  $add1\_L^{time\_cost=12}_{space\_cost=1}$ (

  , $temporary\_am^{time\_cost=0}_{space\_cost=0}$ <temporary tmp0>

  , $disp\_am\_L^{time\_cost=12}_{space\_cost=1}$ <aregister_L a6> <immediate_value_W -4>)

)

  global_cost = $44^{time\_cost=40}_{space\_cost=4}$

- seq (

  $Univ\_assign^{time\_cost=4}_{space\_cost=1}$ (

  $disp\_am\_L^{time\_cost=12}_{space\_cost=1}$ <aregister_L a6> <immediate_value_W -4>

  , $temporary\_am^{time\_cost=0}_{space\_cost=0}$ <temporary tmp0>)

  $add\_L^{time\_cost=6}_{space\_cost=1}$ (

  $disp\_am\_L^{time\_cost=12}_{space\_cost=1}$ <aregister_L a6> <immediate_value_W -8>

  , $temporary\_am^{time\_cost=0}_{space\_cost=0}$ <temporary tmp0>)

  $Univ\_assign^{time\_cost=4}_{space\_cost=1}$ (

  $temporary\_am^{time\_cost=0}_{space\_cost=0}$ <temporary tmp0>

  , $disp\_am\_L^{time\_cost=12}_{space\_cost=1}$ <aregister_L a6> <immediate_value_W -4>)

)

  global_cost = $56^{time\_cost=50}_{space\_cost=6}$

This example shows two derivation chains produced for the previous PMIR.

The first one is produced storing the left son of the PMIR term in the temporary tmp0. Then using the equivalence between the temporay_am access mode and the dreg_am access mode, the pattern matching of the modified PMIR term with the:

add_L (<Dregister_access_L AM1>,<Altmem_access_L AM2>)

succeeds since the disp_am access mode belongs to the Altmem_access_L access class.

The second one is produced storing the left son of the PMIR in the temporary tmp0 first. Then using the equivalence between the temporay_am access mode and the dreg_am access mode, the root and the new left son of the modified PMIR is matched with the:

add_L (<All_access_L AM1>,<Dregister_access_L AM2>)

instruction. Finally, since the destination of the initial PMIR has not been matched a universal store is produced to store the destination of the add_L instruction into the right destination.

Among these two derivation chains, the code selector selects the cheapest one, the cost of which is 44, whereas the second one has a global cost of 56.

2.2.3 SPARC example

Let us consider for the SPARC, the PMIR term that corresponds to the assignment of L := 0. The offset of L from the content of the frame pointer is -4.

```
exec (seq (
    assign_W (
            const_value_13 ( denotation ( value_13. 0 ))
            , designates_address_W (
                    index ( cont_of_gpr_W ( denotation ( gpr. %fp ) )
                            , const_value_13 ( denotation ( value_13. -4 )))))
    ))
```

Three derivation chains are built from the previous SPARC PMIR:

- seq (
  $mov^{time\_cost=4}_{space\_cost=1}$ (

  $value\_13\_am\_L^{time\_cost=2}_{space\_cost=0}$ <value_13 0>

  , $temporary\_am^{time\_cost=1}_{space\_cost=0}$ <temporary tmp0>)

  $Univ\_assign^{time\_cost=5}_{space\_cost=1}$ (

  $temporary\_am^{time\_cost=1}_{space\_cost=0}$ <temporary tmp0>

  , $disp\_am^{time\_cost=1}_{space\_cost=0}$ <gpr_W %fp> <value_13 -4>)
)

  global_cost = $16^{time\_cost=14}_{space\_cost=2}$

- seq (
  $Univ\_assign^{time\_cost=5}_{space\_cost=1}$ (

  $value\_13\_am\_L^{time\_cost=2}_{space\_cost=0}$ <value_13 0>

  , $temporary\_am^{time\_cost=1}_{space\_cost=0}$ <temporary tmp0>)

  $st^{time\_cost=6}_{space\_cost=1}$ (

  $temporary\_am^{time\_cost=1}_{space\_cost=0}$ <temporary tmp0>

  , $disp\_am^{time\_cost=1}_{space\_cost=0}$ <gpr_W %fp> <value_13 -4>)
)

  global_cost = $18^{time\_cost=16}_{space\_cost=2}$

- seq (
  $clr^{time\_cost=4}_{space\_cost=1}$ (

  $disp\_am^{time\_cost=1}_{space\_cost=0}$ <gpr_W %fp> <value_13 -4>)
)

  global_cost = $6^{time\_cost=5}_{space\_cost=1}$

The respective meanings of these three derivation chains produced are respectively store, move and clear.

The first one is produced matching first the root and the left son of the PMIR with the:

mov (<Reg_or_imm_access AM1>, <Gpr_access_W AM2>)

instruction. Moreover, the destination of the initial PMIR hasn't been matched: the disp_am access mode does not belong to the Gpr_access_W access class. So a universal store is produced to store the destination of the move instruction into the right destination. A full description of the pattern matching process can be found in Appendix A.

The second one is got storing the left operand of the PMIR in the temporary tmp0. Then using the access modes declared equivalent to the temporary_am access mode in the interface, this temporary_am access mode can be rewritten into either the gpr_am or int_freg_am access modes. Then, it belongs to the Gpr_access access class. The pattern matching with the:

st (<Gpr_access_W AM1>, <All_access_W AM2>)

instruction succeeds because the disp_am access mode belongs to the All_access_W access class. Here, the temporary resource refers to two kind of registers which are respectively integer register and floating integer register.This last sort of register represents the floating point registers when used to store integers. This points out the use of the binding step after the instruction selection step.

The last one is only produced matching the root and the right son of the PMIR with the:

clr (<All_access_W AM>)

instruction, where the disp_am access mode belongs to the All_access_W access class.

Considering the costs associated with these derivation chains, the instruction selector chooses the cheapest one corresponding to the clear.

This example shows the multiple rewriting possibilities for a PMIR term in spite of its low complexity.

Chapter 3

Binding

3.1 Basic concepts

At the end of the instruction selection step, each subtree of the sequence is a canonical instruction. The leaves of each instruction instance are either actual resources or temporary resources.

In order to prepare the job of the register allocator, it is necessary to bind each temporary access mode with a set of actual access modes. When a temporary is defined by a universal store tree, the related temporary_am access mode is bound to the largest set of constraints deduced from the **Access_class** construct of the interface. This set is restricted when the temporary is used within a following instruction.

Finally, the binder returns a set of available constraints for each temporary. Furthermore, the binder gathers use-def information that are used by the register allocator.

3.2 Input and output of the binder engine

3.2.1 Input term

Let us consider the following result of the instruction selection process in which the LIR instruction appears with its number:

```
seq (
    1   move_L (
            immediate_val_am_L <immediate_value_L 1>
            , disp_am_L <aregister_L a6> <immediate_value_W -4>)
    2   move_L  (
            immediate_val_am_L <immediate_value_L 2>
            , disp_am_L <aregister_L a6> <immediate_value_W -8>)
    3   Univ_assign (
            disp_am_B <aregister_L a6> <immediate_value_W -12>
            , temporary_am <temporary tmp0> )
    4   addi_W (
            immediate_val_am_W <immediate_value_W 24>
            , temporary_am <temporary tmp0> )
    5   Univ_assign (
            disp_am_L <aregister_L a6> <immediate_value_L 12>
            , temporary_am <temporary tmp1>)
    6   Univ_assign (
```

```
        temporary_am <temporary tmp0>
        , temporary_am <temporary tmp2>)
7   muls (
        disp_am_W <aregister_L a6> <immediate_value_W -4>
        , temporary_am <temporary tmp2> )
8   Univ_assign (
        temporary_am <temporary tmp2>
        , areg_ind_am_L <aregister_L tmp1> ))
```

3.2.2 Output tables

3.2.2.1 Binding constraints

This engine gathers the constraints on resources, i.e the the access modes that must be related to each of them. When a temporary tmpi is defined by a universal assignment of the form: Univ_assign (..., temporary_am (temporary tmpi)) , it gets the maximal degree of freedom for binding, i.e it can be bound to all the access modes of the interface. This set is generally restricted when the temporary is used in a following instruction. Let us consider instruction number 4, it is a "addi_W" instruction :

```
addi_W ( immediate_val_am_W <immediate_value_W 24>
       , temporary_am <temporary tmp0>)
```

It is an instance of the canonical form :

addi_W (<Immediate_access_W ...>), <Dregister_access_W ...>)

Thus temporary_am access mode of instruction 4 must belong to the Dregister_access_W class, it means that the right access mode allowed for tmp0 is dreg_am_W.

tmp1 is bound to a aregister_L because it appears in an areg_ind_am access mode in instruction 8.

tmp2 is bound to dreg_am_W before instruction 7 and to dreg_am_L in instruction 7 and after. This is set according to the template of muls that expresses that muls performs a multiplication on two word operands and puts the result as a long word in the second operand.

Finally the following constraints are built by the binder :

First a table that sets the list of resource sorts allowed for a temporary by means of its direct addressing mode.

```
tmp0  =  { dreg_am_W }.
tmp1  =  { areg_am_L }.
tmp2  =  { dreg_am_L }.
```

Second, a table that gives the size of a temporary in each instruction where it appears.

```
tmp0  =  [ [3,W],[4,W],[6,W]].
tmp1  =  [ [5,L],[8,L]].
tmp2  =  [ [6,W],[7,L],[8,L]].
```

When tmp0 is defined in instruction 3, its size is undefined, but it takes the size "W" in instruction "addi". Similarly when tmp2 is defined in instruction 6, its size is undefined but it

takes the size "W" as an operand of the instruction muls (according to the specification of section 1.8.1); i.e before the execution of the "muls" instruction. Since the size of the destination operand of the muls instruction is "L", tmp2 takes this size from instruction 7. Finally the size of tmp1 is undefined from instruction 5, it will take the size "L" because it is bound to a aregister_L storage class in instruction 8.

3.2.2.2 Usedef tables

A use-def table has one entry per temporary. An entry for a temporary is a list of triples. A triple includes three items of information :

- the first one is the number of the instruction in which the temporary appears.
- the second one is 1 if the temporary is used in the instruction, 0 otherwise.
- the third one is 1 if the temporary is defined in the instruction, 0 otherwise.

```
tmp0  =  [ [3,0,1],[4,1,1],[6,1,0]].
tmp1  =  [ [5,0,1],[8,1,0]].
tmp2  =  [ [6,0,1],[7,1,1],[8,1,0]].
```

In the instruction number 3, tmp0 is defined since it is in destination position of a Univ_assign. In the instruction number 4, tmp0 is used and defined in instruction 4 since it is the binary arithmetic instruction addi_W.

Chapter 4

The register allocator

The input of the register allocator includes temporaries and universal store instructions. The register allocator works in two steps :

1. All the temporaries generated by the instruction selector or those which are already in the PMIR must be allocated, i.e the infinite set of temporaries must be restricted to the finite set of available registers of the target machine. When no more registers are available, the register allocator must choose a register and spill it in memory, that is commonly called spill code generation.

2. The universal store instructions must be rewritten into actual instructions of the LIR according to the colors assigned to the temporaries. In the sequel, we denote "univ_assign_rewriting" this step.

Finally, the output of the register allocator is a sequence of instructions in their canonical form for which assembly code can be emitted.

4.1 Temporary coloring and spill code production

4.1.1 Basic concepts

A global register allocator achieves these goals. It is currently performed by an algorithm that is an adaptation of the CHOW and HENNESSY algorithm [CH 90]. A register allocator generated by PAGODE uses a data-flow analysis of the LIR, the binding tables on temporaries, the description of the physical resources deduced from the target machine specification and an additional specification for the production of spill code. In this algorithm, the live range of a variable is defined as the set of basic blocks where this variable is live. A variable is colored to a single register inside a live range. During the register allocation, a live range can be split into two "sub" live ranges in order to decrease the number of coloring constraints on this live range. Therefore, a variable can be colored to different registers depending on the "sub" live ranges considered.

The concept of interfering live-ranges of variables of CHOW and HENNESSY has been extended to a more fine grain definition for temporaries that are alive in a single basic block.

Optimizations before code selection can introduce temporaries that are global to the whole program or local to a procedure. The instruction selector of PAGODE produces temporaries that are always local to a basic block. The register allocator is intra-procedural, it is able to handle

temporaries produced by the instruction selector as well as the temporaries that already exist in the PMIR.

Spill strategies currently available spill temporaries into a stack or into an area located in the load module at the end of the procedure. In the current state of implementation, no specific spill strategies depending on the temporary scope are available. The choice of a given strategy holds for all the temporaries. In the future, the PAGODE system will be able to apply a distinct strategy for each kind of temporaries when a symbol table for temporaries will be inherited from the front end.

Currently, the register allocator has to build the control flow graph and split the program into procedures. A procedure is supposed to have a single entry point and a single exit point.

The set of procedure names is built from the labels occuring in the instructions of the program whose **$modification** value is "call_proc".

The exit point of a procedure is the first instruction that has "return_proc" as value of **$modification**. All the following instructions that have "directive" as value of **$modification** are expected to belong to the procedure. In particular, the register allocator is expected to find the directive giving the size of the stack among these instructions (when there is an instruction that sets the stack in the procedure).

4.1.2 The target machine specification

4.1.2.1 Register declarations

The description of the available registers is automatically deduced from the storage base specification in the target machine specification. More precisely, for each storage class related to registers, the following properties are given as input to the register allocator:

- the storage base,
- the access mode to access directly this storage class,
- the size in bytes (the size must be available in order to spill the right amount of bytes in the stack or in the area),
- single register or register pair,
- the actual names of the registers of this storage class.

```
Storage_base DREG                          - - Data registers
    Symbolic_notation
            'DREG' is d(0..7)
    $base_units  =   4                     - - Maximal number of units
    $ free       =   %d(0..7)
End


Storage_class dregister
    Denotation < dregister!size reg>
    Attributes
        $Base = DREG
    ...
    ...
    Instances
```

```
        size in {B, W, L}
        case size is
            B   :   size =  1 base_units
            W   :   size =  4 base_units
            L   :   size =  4 double base_units
        End case
End
```

The input of the register allocator can be either the output of the binder engine for CISC machines or the output of the first pass of scheduling for RISC machines (see chapter 6). Register allocation on the previous example (see section 3.3) gives the following output :

```
seq (
    1   move_L (
            immediate_val_am_L <immediate_value_L 1>
            , disp_am_L <aregister_L a6> <immediate_value_W -4>)
    2   move_L  (
            immediate_val_am_L <immediate_value_L 2>
            , disp_am_L <aregister_L a6> <immediate_value_W -8>)
    3   Univ_assign (
            disp_am_B <aregister_L a6> <immediate_value_W -12>
            , dreg_am_W <dregister_W d0> )
    4   addi_W (
            immediate_val_am_W <immediate_value_W 24>
            , dreg_am_W <dregister_W d0> )
    5   Univ_assign (
            disp_am_L <aregister_L a6> <immediate_value_L 12>
            , areg_am_L <aregister_L a0>)
    6
    7   muls (
            disp_am_W <aregister_L a6> <immediate_value_W -4>
            , dreg_am_L <dregister_L d0> )
    8   Univ_assign (
            , dreg_am_L <dregister_L d0> )
            , areg_ind_am_L <aregister_L a0> ))
```

Instruction 6 does not exist any longer because it is a useless copy from d0 with size W to d0 with size W.

4.1.2.2 Spill code strategies and target machine specification

In order to generate the appropriate spill code, the register allocator uses the **spill code** construct that depicts a particular computation needed by a given spill code strategy.

Two spill code strategies must be specified :

- the spill in stack strategy : the contents of a register is spilled on the stack, it is introduced by the keyword **spill_in_stack** :
- the spill in area strategy : the contents of a register is spilled into an area located in the load module at the end of the procedure, it is introduced by the keyword **spill_in_area**.

The spill code specification includes two sections, one for each of these strategies. These two sections begin with the definition of the variables of the specification denoting the values of interest.

4.1.2.3 Spill in stack

The main hypotheses of this strategy are:

- There exists a fixed base and a variable offset that depends on each temporary to access a cell on the stack. A specific access mode is declared that performs this access.
- Specific load and store instructions depending on the storage class of the register are listed.
- At the end of each procedure, there is a directive that gives the equivalence between a label and the stack size.
- The stack is allocated at the beginning of the execution of the procedure using the label variable.

First of all, the compiler writer must declare the names of the variables denoting the offset, label, register and stack_size values.

Then he must specify the access mode used to describe the access path to a cell of a stack using the offset variable is specified. For instance:

disp_am!size (<aregister_L a6> , <immediate_value_W X>)

The **load** keyword begins a construct where the instructions that can be used to load the contents of a cell of the stack in a given register are specified. There must be one load instruction specification per parameterized storage class of register.

Secondly, the **store** construct depicts the instructions that can be used to store the contents of a given register in a cell of the stack. The PAGODE constructor checks the consistency between the load and the store parts declarations.

The spill code generator works under the hypothesis that a spill instruction is always specified giving the parameters in the following order: first the source operands then the destination operand.

The **stack_link** construct declares the instruction used to load the stack base.

The **def_stack_size** construct describes the directive that fixes the size of the stack for a procedure.

```
Spill code
    case spill_in_stack :
        {
        offset : X;
        label : LAB;
        register : REG;
        stack_size : STACK_SIZE;
        Reserved registers : < aregister a6>
        Access_mode :
            disp_am!size (<aregister_L a6> , <immediate_value_W X>)
        load :
            move!size (disp_am!size (<aregister_L a6>, <immediate_value_W X>)
                    , dreg_am!size (<dregister!size REG>))
                    where size in { B, W, L }
            move!size (disp_am!size (<aregister_L a6>, <immediate_value_W X>)
                    , areg_am!size (<aregister!size REG>))
                    where size in { W, L }
        store :
            move!size (dreg_am!size (<dregister!size REG>))
                    , disp_am!size (<aregister_L a6>, <immediate_value_W X>)
```

```
                where size in { B, W, L }
        move!size (areg_am!size (<aregister!size REG>))
                , disp_am!size (<aregister_L a6>, <immediate_value_W X>)
                where size in { W, L }
    stack_link :
        link (areg_am_L (<aregister_L a6>)
            , immediate_dlab_am_W (<data_label_W LAB>))
    def_stack_size :
        equ (immediate_dlab_am_W (<data_label_W LAB>)
            , immediate_val_am (<immediate_value_W STACK_SIZE>))
    Resource_length
        case size is
            B : 1
            W : 4
            L : 8
        End case
    }
case spill_in_area :
    { ...
```

An example that has been run with the two stategies one after the other can be found in appendix B. In order to demontrate the results of spill code generation, the number of available data registers has been reduced to d0 and d1.

4.2 Universal store rewriting

4.2.1 Basic concepts

At the end of the previous step, all instructions that are not universal stores match exactly the target machine specification. Owing to the previous step, the operands of the universal stores are actual access modes on actual resources.

The remaining task consists in finding an instruction of the spill or load or store families that matches the operands. If such an instruction exists, the universal store operator is rewritten according to this instruction. Otherwise, the PAGODE system assumes that there is a size misfit. In some cases, there exist assembly instructions that perform size conversions. This must be specified by the compiler writer in a special purpose construct called **Convert declaration**. If such a misfit can be solved by a conversion instruction, this instruction is inserted. By default, the PAGODE system assumes that the implicit conversion is performed by the assembler and the operand that will be converted is chosen : it is an operand that does not refer to the memory. The size of that operand is modified in order to match an instruction.

4.2.2 Target machine specification

4.2.3 The convert constructor

A convert declaration specifies a list of conversions. An item of this list associates a conversion between two access modes that only differ on their sizes and the instruction to generate when this conversion is applied.

Let us consider the conversion instructions on the MC68000:

Convert declaration

```
areg_am_W (<aregister_W X>) -> areg_am_L (<aregister_L X>)
            : movea_W (areg_am_W(<aregister_W X>)
                       , areg_am_L (<aregister_L X>))
dreg_am_W (<dregister_W X>) -> dreg_am_L (<dregister_L X>)
            : ext_L (dreg_am_L (<dregister_L X>))
dreg_am_B (<dregister_B X>) -> dreg_am_W (<dregister_W X>)
            : ext_W (dreg_am_W (<dregister_W X>))
```

End Convert

4.2.4 Error messages

The PAGODE constructor checks whether an access mode occurring in an item of the convert declaration has been previously declared. When this condition does not hold the following error message is emitted :

- Error : Unknown access mode ...in the Convert construct.

It also checks whether an instruction occurring in an item of the convert declaration has been previously declared with the "convert" value as modification. When this condition does not hold the following error message is emitted :

- Error : The instruction ...used in the Convert construct must have been defined with the "convert" value as modification attribute.

The following result is obtained of the term of the section 4.1.2.1 :

```
seq (
    1   move_L (
            immediate_val_am_L <immediate_value_L 1>
            , disp_am_L <aregister_L a6> <immediate_value_W -4>)
    2   move_L  (
            immediate_val_am_L <immediate_value_L 2>
            , disp_am_L <aregister_L a6> <immediate_value_W -8>)
    3   move_B (
            disp_am_B <aregister_L a6> <immediate_value_W -12>
            , dreg_am_B <dregister_B d0> )
    3'  ext_W (
            , dreg_am_W <dregister_W d0> )
    4   addi_W (
            immediate_val_am_W <immediate_value_W 24>
            , dreg_am_W <dregister_W d0> )
    5   movea_L (
            disp_am_L <aregister_L a6> <immediate_value_L 12>
            , areg_am_L <aregister_L a0>)
    6
    7   muls (
            disp_am_W <aregister_L a6> <immediate_value_W -4>
            , dreg_am_L <dregister_L d0> )
    8   move_L (
            , dreg_am_L <dregister_L d0> )
            , areg_ind_am_L <aregister_L a0> ))
```

Using the ext_W instruction of the convert specification, the third instruction is rewritten into the sequence of 3 and 3'. The fifth instruction of root Univ_assign is rewritten into a movea instruction since the destination is an address register.

Chapter 5

The scheduler

Two kinds of parallelism exist inside processors. There exist either superscalar or pipelined architectures : the superscalar can issue several instructions during the same cycle ; the pipeline overlaps instruction execution. The PAGODE system has been designed to generate a scheduler for pipelined machines.

5.1 Basic concepts

5.1.1 Pipeline technique

Pipelining is an implementation technique whereby multiple instructions overlap during execution. Each step in the pipeline completes a part of an instruction. Each of these steps is called a pipe stage.

There are situations called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. In that case, a hardware interlock occurs, i.e. the execution of the next instruction is suspended. There are three classes of *hazards* :

1. Data hazards arise when the second instruction needs a data not yet computed by the first one.

2. Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of pairs of instructions in simultaneous overlapped execution.

3. Control hazards arise from dependencies on branches and other instructions that change the program counter.

Throughout this chapter, examples are taken from the SPARC CYPRESS specification.

5.1.2 Short overview of the PAGODE scheduler

Before scheduling, an instruction sequence is available. This sequence may generate various hazards and doesn't take advantage of all the machine abilities. To optimize execution speed, the scheduler must focus on the arrangement of the instructions in order to decrease the number of hazards.

The current scheduler of PAGODE runs on each basic block. It is an adaptation of a list-scheduler. It relies on the Gibbons and Muchnik algorithm [GM 86] which gives a cover of the "data dependence graph". The data dependence graph is built from the initial sequence. In that

graph the nodes represent the instructions, the edges are decorated with the minimal number of cycles such that the second instruction can be issued after the first without data hazards. Our heuristic gives the highest priority to the instruction whose successors in the graph have a hazard. The chosen instruction must have numerous successors.

A new sequence that must comply with the old semantic is built from a topological sort of the data dependence graph in order to obey precedence data constraints. Instructions are considered one by one and added to the end of the new sequence when no structural hazards with previously issued instructions exists. If no instruction satisfies this condition, a nop is inserted.

Branch hazards are avoided by filling up the wait cycles with independent instructions provided by the current block or one of the following blocks.

Pipeline management is entirely done by the PAGODE scheduler. This means that nop instructions are generated to avoid interlocks.

5.2 The target machine specification

5.2.1 Specifications related to data hazards

There are three classes of data hazards : RAW (*Read After Write*), WAW (*Write After Write*), and WAR (*Write After Read*). In fact the RAW hazards are the most frequent on our target machines. The computer companies have encouraged the integration of bypasses. Bypass is a hardware mechanism for reducing the number of RAW hazards. The second instruction can catch the result before the first one completes. In PAGODE , this mechanism is represented by a global table called **Delay_table**. An array element of this table is the minimal delay that must be satisfied to avoid this hazard between two instructions ins_1 and ins_2.

Let us consider two instructions ins_1 and ins_2 such that :

- ins_2 follows ins_1 in the semantic order of execution
- reg is a register that is written in ins_1 and read in ins_2.

register	use in ALU	use in address	use in store	use in cc
result of LOAD	2	2	2	2
result of ALU	1	1	2	1
side-effect	0	0	0	0

The delay table has three lines that correspond to the kind of write access of reg in ins_1 :

1. the value written in the register reg is the result of a load.
2. the value written in the register reg is the result of an arithmetic or logical computation.
3. the value written in the register reg is the result of an auto-incrementation[1].

The delay table has four columns that correspond to the kinds of read access of reg in ins_2 :

1. the value read in the register reg is used for an arithmetic or logical computation.
2. the value read in the register reg is used for an address computation.

[1] i.e. A pre- or post-incrementation

3. the value read in the register reg is used for a store in the memory.

4. the value read in the register reg is used for a condition code computation.

Delay_table = ((2,2,2,2) (1,1,**2**,1) (0,0,0,0))

The value **2** means that ins_1 is an arithmetic instruction which writes the register reg and ins_2 a store of the contents of reg in memory and that ins_2 must begin at least **2** cycles after ins_1. Since the delay-table has constant dimensions and since there exists no auto-incrementation addressing mode on the SPARC, the last four "0" stand for undefined values.

Some instructions do not need any table searching because their execution is longer and without bypass. Table searching is only done when the boolean **$use_delay_table** is set to TRUE on a specific instruction. The default value is FALSE. So the delay-table is done only for some RAW data hazards.

In order to deal with other hazards RAW, WAW and WAR, the code generator writer must use two attributes available on each instruction that is not a branch one:

- The stage number where the source operands are read is denoted by the attribute **$init_read_cycle**. On a specific instruction if this action needs more than one stage, we consider the first stage of this step and we assume that fetching all the source registers begins at the same time.

- **$end_write_cycle** denotes the stage number where the result operand is written. If on a specific instruction this action needs more than one stage, we take the last stage of this step.

In order to deal with branch instructions, we assume that the read stage is the same for all branches. The value of the global variable **Branch_init_read_cycle** is the stage number where the operands of the branch instructions are read.

The scheduler must be able to deal with target machines like i860 using auto-incrementation addressing modes where the write-back step does not execute at the same time as the side-effect stage; we assume that all the instructions using side-effect addressing modes have the same side-effect stage.

If an instruction uses a register with an auto-incrementation, the boolean **$side_effect_OK** is set to TRUE. In that case the scheduler looks into the **Delay_table** to avoid RAW data hazards on the register in auto-incrementation. WAR and WAW hazards can be managed using the value of the side effect stage which is denoted by the attribute **$side_effect_write_cycle** on each related instruction.

Two examples follow.

- addition instruction:

```
Instruction
    Canonical_form
        add (<Gpr_access_W AM1>, <Reg_or_imm_access AM2>
                , <Gpr_access_W AM3>)
    Attributes
        ...
        $end_write_cycle   =   4
        $init_read_cycle   =   2
        $use_delay_table   =   true
        ...
    Template
        assign_W (add_32 ...
End
```

- Floating-point instruction : fdivd

Instruction
Canonical_form
fdivd (<Freg_access_D AM1>, <Freg_access_D AM2>
, <Freg_access_D AM3>)
Attributes
...
$end_write_cycle = 41
$init_read_cycle = 3
...
Template
assign_D (fdivd ...
End

The lack of specification of **$use_delay_table** in fdivd means that the value is FALSE and that it is not necessary to search in the **Delay_table**.

5.2.2 Specifications related to structural hazards

These hazards are managed by means of a booking table. It is necessary to list all the functional units which are components of the processor and to specify the maximal number of uses in the same clock cycle. We assume that the **Interlock_table** is available. It includes the list of all functional units which are used by the instruction set processor. The maximal number of uses without structural hazard is related to each of them.

For instance, on the SPARC, let us denote DATA_BUS and ADR_BUS the data bus and the address bus. READ_REG_PORT and WRITE_REG_PORT denote the read port and the write port on the *register file*. ALU, SU, FALU, et FMDCSU denote respectively the Arithmetic and Logic Unit, the Shift Unit, the Floating point Arithmetic and Logic Unit, the Floating point Multiply Divide Compare and Square-root Unit. PSR and FSR are respectively the Processor Status Register and the Floating point Status Register.

The interlock table specification is the following for the SPARC :

Interlock_table

```
DATA_BUS : 2
ADR_BUS : 1
READ_REG_PORT : 1
WRITE_REG_PORT : 1
DECODE : 1
ALU : 1
SU : 1
FALU : 2
FMDCSU : 2
PSR : 1
FSR : 1
PROGRAM_COUNTER : 1
```

A table called **$Booking_table** is available for each instruction. It includes the list of stage numbers which are necessary for the execution of the instruction. For each of them, the functional units which are used are merely listed using the same formalism as in the **Interlock_table**. A stage

which uses no fonctional unit is only numbered and kept empty. The stage number may be followed by an asterisk(*), it means that in the execution context of the instruction a shift of one or more cycles can be introduced. It will be explained in the following.

For instance in the following **$Booking_table** attribute, the first stage uses both ADR_BUS and DATA_BUS. The second stage uses both DECODE and READ_REG_PORT.

```
$Booking_table =    (1 : ADR_BUS / / DATA_BUS ;
                     2* : DECODE / / READ_REG_PORT ;
                     3* : ALU ; 4* : WRITE_REG_PORT / /
                     ADR_BUS / / DATA_BUS)
```

A critical point to detect structural hazards is to connect the stages of instructions in the pipe during execution. The Cypress manufacturer implements a specific hardware system of pipeline execution using the Internal Op-code feature (i.e : IOP). It is a hardware mechanism that handles multi-cycle instructions. The aim is to differ data bus access for either loading a data or loading an instruction to execute. Multi-cycle instructions are usually loads and stores. They induce a shift of one or more clock cycles in a part of stages of some following instructions.

On the SPARC architecture, there exist three kinds of multi-cycle instructions. They define the number of IOPs on each of them and the behaviour of each of them :

- A double-cycle instruction has only one IOP, i.e the two instructions that follow it are shifted of one cycle. That is the case of the "ldsb" specified below. "single loads", "jump" and "Return from Trap" belong to the family of double cycle instructions. Their behaviour in the pipe is the following. Letters denotes instructions and numbers denotes stage numbers. The notation *iop(A)* means that the cycle is used by the IOP of the double-cycle instruction A.

instr A	**A-1**	**A-2**	**A-3**	**A-4**				
instr B		B-1	*iop(A)*	*iop(A)*	*iop(A)*			
instr C			C-1	B-2	B-3	B-4		
				A-5	C-2	C-3	C-4	
instr D					D-1	D-2	D-3	D-4

  The example shows the 5 cycles of A, the shift of 1 inside B and C, and the normal execution of D.

- A triple-cycle instruction has two IOP, i.e the two instructions that follow it are shifted of two cycles. "single stores" and "double loads" belong to that family.

- A quadruple-cycle instruction has three IOP, i.e the two instructions that follow it are shifted of three cycles. "double stores" and "atomic Load-Store" belong to that family.

The SCALA language provides two attributes to take those features into account :

- **$IOP_cycle_shift** is the clock shift number on the following instructions.

- **$IOP_patched_ins** is the number of following instructions that come under this shift.

Notice that the sequence of instructions producing IOPs induce a cascade of shifts that the scheduler must manage. The main issue is that the IOPs are only known at the decode stage, and that stage

itself can be under a previous shift. The number of IOPs is currently limited to three IOPs per instruction, since we have not encountered more sophisticated cases in the target machines we modeled.

The following specification is a whole instruction. It is a double-cycle instruction.

```
Instruction
    Canonical_form
        ldsb (<Gpr_access_W AM1>, <Gpr_access_W AM2>)
    Attributes
        $length =   W
        $fmt    =   ~ ldsb $fmt (<Gpr_access_W AM1>)
                          ,$fmt (<Gpr_access_W AM2>) ~
        $Booking_table          =   (1 : ADR_BUS / / DATA_BUS ;
                                     2* : DECODE / / READ_REG_PORT ;
                                     3* : ALU ; 4* : WRITE_REG_PORT / /
                                     ADR_BUS / / DATA_BUS)
        $IOP_cycle_shift        =   1
        $IOP_patched_ins        =   2
        $end_write_cycle        =   4
        $init_read_cycle        =   2
        $use_delay_table        =   true
    Template
        assign_W (sign_extend (right_justify (src (<All_access_W AM1>)))
                  . dst (<Gpr_access_W AM2>))
End
```

The second and the following stages denoted with an asterisk can be shifted by a previous IOP.

5.2.3 SPARC declaration examples

The various features dealing with data hazards and structural hazards can be summarized by the "add" instruction on general purpose register and the floating-point divide instruction.

- addition instruction :

```
Instruction
    Canonical_form
        add (<Gpr_access_W AM1>, <Reg_or_imm_access AM2>
             , <Gpr_access_W AM3>)
    Attributes
        ...
        $Booking_table      =   (1: ADR_BUS / / DATA_BUS ;
                                 2* : DECODE / / READ_REG_PORT ;
                                 3* : ALU ; 4* : WRITE_REG_PORT)
        $end_write_cycle    =   4
        $init_read_cycle    =   2
        $use_delay_table    =   true
    Template
        assign_W (add_32 ...
End
```

- Floating-point instruction : fdivd

```
Instruction
    Canonical_form
        fdivd (<Freg_access_D AM1>, <Freg_access_D AM2>
                , <Freg_access_D AM3>)
    Attributes
        ...
        $Booking_table      =   (1 : ADR_BUS / / DATA_BUS ;
                                 2* : DECODE ; 3* : READ_REG_PORT ;
                                 4* : READ_REG_PORT ;
                                  5* - > 40* : FMDCSU ;
                                 41* : FMDCSU / / WRITE_REG_PORT)
        $end_write_cycle    =   41
        $init_read_cycle    =   3
    Template
        assign_D (fdivd ...
End
```

The stages going from 5 to 40 are all similar.

5.2.4 Specifications related to control hazards

A control hazard arises when a branch updates the program counter with a new address and the processor must wait for it to fetch the next instruction. This delay is called branch slot. In order to fill up this time with independent instructions, it is necessary to know the size of the branch slot. The optimizer can be more efficient if the scheduler knows whether the slot instructions will be executed whether the branch is taken or not. We assume that the two following data on branches are available :

- The size in number of instructions of the slot is called **$delay_slot**. In most cases the value is 1.
- The behaviour of the instructions which fill up the slot is depicted by **$executed_slot**. Four values are allowed :
    1. *always*: the instructions are always executed. This case happens on all machines for unconditional branches.
    2. *taken*: the instructions are only executed when the branch is taken. This case can occur on conditionals branches and is nonsense for unconditional branches. It occurs on the SPARC.
    3. *untaken*: the instructions are only executed when the branch is not taken. This case can occur on conditionals branches and is nonsense for unconditional branches. It occurs on the i860, M88000 but not on the SPARC.
    4. *never*: the instructions are never executed. This case happens rarely and in very specific situations. It may occur for the SPARC.

The branch instruction "bne,a" with "annul" bit field "a" can be specified as follows:

```
Instruction
    Canonical_form
        bne (<Relative_access AM>)
    Attributes
    $length =   W
    $fmt    =   ~ bne,$fmt (<Relative_access AM>) ~
    $Booking_table      =   (1 : ADR_BUS / / DATA_BUS ;
                             2* : DECODE / / PSR / / PROGRAM_COUNTER)
    $executed_slot      =   taken
    $delay_slot         =   1
Template
    branch_if_ne (dst(<Relative_access AM >))
End
```

5.3 Input and output of the scheduler engine

Let us consider the following LIR term:

```
seq (
    1   ld ( disp_am <gpr_W %fp> <value_13 -16>
             , temporary_am <temporary tmp1> )
    2   ld ( disp_am <gpr_W %fp> <value_13 -4>
             , temporary_am <temporary tmp2>)
    3   add ( temporary_am <temporary tmp1>
             , temporary_am <temporary tmp2>
             , temporary_am <temporary tmp3>)
    4   add ( temporary_am <temporary tmp3>
             , temporary_am <temporary tmp1>
             , temporary_am <temporary tmp4>)
    5   st ( temporary_am <temporary tmp4>
             , disp_am <gpr_W %fp> <value_13 -32>)
    6   cmp ( temporary_am <temporary tmp1>
             , temporary_am <temporary tmp2>)
    7   bg ( relative_am <code_label l1>)
    )
```

A data dependence graph the nodes of which are the various tmp_i, the actual registers that belong to the LIR (here fp) and a constant node called "MEM" for memory references is built from the LIR and given as input to the list-scheduling processor.

The result of the scheduling is the following:

```
seq (
    1   ld ( disp_am <gpr_W %fp> <value_13 -16>
             , temporary_am <temporary tmp1> )
    2   ld ( disp_am <gpr_W %fp> <value_13 -4>
             , temporary_am <temporary tmp2>)
    3   add ( temporary_am <temporary tmp1>
             , temporary_am <temporary tmp2>
             , temporary_am <temporary tmp3>)
    4   add ( temporary_am <temporary tmp3>
             , temporary_am <temporary tmp1>
             , temporary_am <temporary tmp4>)
    6   cmp ( temporary_am <temporary tmp1>
```

```
        , temporary_am <temporary tmp2>)
5   st ( temporary_am <temporary tmp4>
        , disp_am <gpr_W %fp> <value_13 -32>)
7   bg ( relative_am <code_label l1>)
)
```

The following figure shows what happens in the pipe. Let us denote I, II, III, IV, V and VI the cycle stages. There exists a delay between the instructions 4 and 5 in the data dependence graph.

The following two dimension table represents pipeline usages. Lines stand for instruction issues and columns stand for clock cycles.

Cycle	1	2	3	4	5	6	7	8	9
ins 1	**1-I**	**1-II**	**1-III**	**1-IV**					
ins 2		2-I	*iop(1)*	*iop(1)*	*iop(1)*				
ins 3			3-I	2-II	2-III	2-IV			
				1-V	*iop(2)*	*iop(2)*	*iop(2)*		
ins 4					4-I	3-II	3-III	3-IV	
						2-V	4-II	4-III	4-IV
ins 6							6-I	6-II	6-III

Cycle	7	8	9	10	11	12	13	14	15
ins 6	*6-I*	*6-II*	*6-III*	6-IV					
ins 5		5-I	5-II	5-III	5-IV				
ins 7			7-I	*iop1(5)*	*iop1(5)*	*iop1(5)*			
empty				x	*iop2(5)*	*iop2(5)*	*iop2(5)*		
					5-V	7-II	7-III	7-IV	
						5-VI	x	x	x

The 5-II stage ought to be executed after or at the same time than the 4-IV. The insertion of the instruction 6 avoids a nop.

Chapter 6

Overall structure of the code generators produced

6.1 Code generators for CISC machines

A CISC code generator runs sequentially, performing the following steps according to the figure 6.1 :

1. the instruction selection.
2. binding constraints are built by the binder engine.
3. then register allocation is performed.
4. the Univ_assign operator is overloaded by the "assign_rewriter" engine.
5. the code is emitted.

6.2 Code generators for RISC machines

6.2.1 Integration of register allocation and scheduling

The interaction between register allocation and scheduling is yet a research topic. One main feature of this interaction is that scheduling change the live range of resources.

Basically, we choose to integrate these two engines according to the following basic framework according to the figure 6.2 :

1. A *first pass of scheduling* works on the term got as output of the prepare engine after the instruction selection step.
2. Then *register allocation* is performed.
3. Since the register allocation produces spill code, i.e new sequences that have not been dealt by the scheduler, a *second scheduling pass* is necessary.

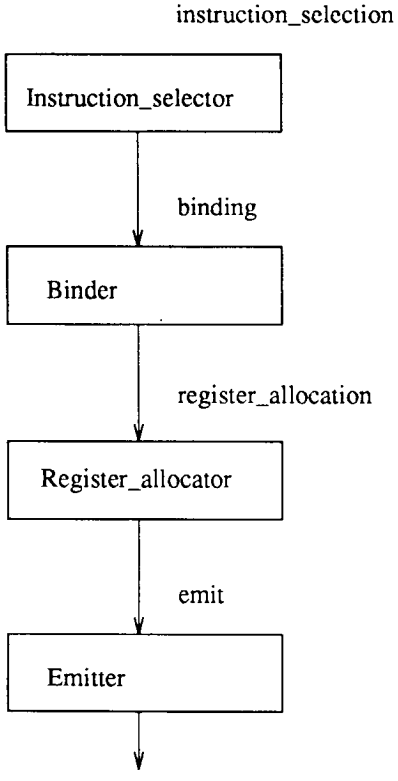


Figure 6.1: The overall structure of a CISC back-end.

At the end of the binding step, Univ_assign operators and temporay_am access modes occur yet in the term. Basically they correspond to several degrees of freedom for the corresponding access modes. A main hypothesis of the PAGODE system is to keep these degrees of freedom as long as possible, and especially until the register allocator so that this last engine can take advantage of the binding sets.

A main consequence of this choice is that no scheduling can be done on the Univ_assign operators, because they cannot be related to any load, store or move instruction. That is the reason why three pseudo-instructions must be specified by the compiler writer, they must be called **Univ_load**, **Univ_store**, **Univ_move**, that are three keywords of the PAGODE system.

These three pseudo-instructions are specified as true instructions such that the behaviour of the Univ_load is the average behaviour of the loads of the instruction set processor from the point of view of the pipelined execution. The Univ_store is specified similarly regarding the stores of the instruction set processor. Finally, the Univ_move is specified regarding the moves of the instruction set processor.

Thus, the prepare engine rewrites the Univ_assign operators into either Univ_load or Univ_store or Univ_move. This overloading mechanism strongly relies on the fact that a temporary_am access mode depicts only an access to a resource. It also uses an information built by the PAGODE constructor that tells wether or not an access mode is a memory reference. Since the first operand of an Univ_assign is the source operand and the second one the destination operand, the following algorithm is applied :

- if the first operand is a memory reference the Univ_assign operator is overloaded by a

Univ_load,

- if the second operand is a memory reference the Univ_assign operator is overloaded by a Univ_store,
- otherwise, the Univ_assign operator is overloaded by a Univ_move.

The prepare engine is then called after the instruction selector and before the first pass of scheduling.

Currently, the code generators produced by the PAGODE system run sequentially performing the following steps after instruction selection :

1. the prepare engine overloads the Univ_assign operators of the term got from the instruction selector.
2. a first pass of scheduling works on the term got from the prepare engine.
3. the binding step is performed.
4. the register allocation is performed.
5. the Univ_load, Univ_store and Univ_move operators are overloaded by true operators of the instruction set processor by a special purpose engine called "assign_rewriter".
6. Since the register allocation produces spill code, i.e new sequences that have not been dealt by the scheduler, a new scheduling pass is necessary.
7. the code is emitted by the emit engine.

6.2.2 Specification for integration

Three instruction specifications related to Univ_load, Univ_store, Univ_move must exist in the target machine specification.

The Univ_load specification is an instruction which has the average behaviour of the instructions of the "load" family. It uses the average length of the pipe. Thus for the SPARC architecture, this specification is the same as that of the "ldd" instruction and for the rest it gets the specifications of the "ld" instruction.

```
Instruction
    Canonical_form
        Univ_load (<All_access_W AM1>, <Gpr_access_W AM2>)
    Attributes
        $length =   {B, H, W, D}
        $fmt    =   ~ ld $fmt(<All_access_W AM1>)
                         ,$fmt(<Gpr_access_W AM2>) ~
        $Booking_table      =   (1 : ADR_BUS / / DATA_BUS ;
                                 2* : DECODE / / READ_REG_PORT ;
                                 3* : : 4* : WRITE_REG_PORT / /
                                 ADR_BUS / / DATA_BUS
                                 5 : ADR_BUS / / DATA_BUS)
        $IOP_cycle_shift    =   2
```

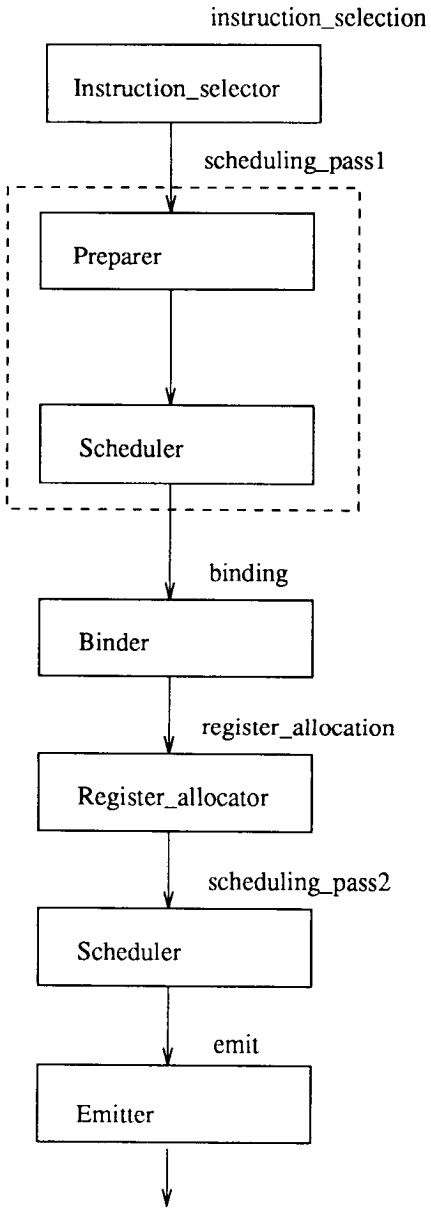


Figure 6.2: The overall structure of a RISC back-end.

```
        $IOP_patched_ins      =   2
        $end_write_cycle      =   5*
        $init_read_cycle      =   2
        $use_delay_table      =   true
    Template
        assign_W (src( <All_access_W AM1>)
                  , extend_long (dst (<Gpr_access_W AM2>)))
End
```

Similarly, the Univ_store specification is a copy of the "st" instruction with pipe attributes that are those of the "std" instruction.

```
Instruction
    Canonical_form
        Univ_store (<Gpr_access_W AM1>, <All_access_W AM2>)
    Attributes
        $length =   {B, H, W, D}
        $fmt    =   ~ st  $fmt (<Gpr_access_W AM1>)
                          ,$fmt  (<All_access_W AM2>) ~
        $Booking_table        =   (1 : ADR_BUS / / DATA_BUS ;
                                   2* : DECODE / / READ_REG_PORT ;
                                   3* : READ_REG_PORT ;
                                   4* : ; 6* : ADR_BUS / / DATA_BUS/
        $IOP_cycle_shift      =   3
        $IOP_patched_ins      =   2
        $end_write_cycle      =   6*
        $init_read_cycle      =   2
        $use_delay_table      =   true
    Template
        assign_W (src (<Gpr_access_W AM1>)
                  , (dst (<All_access_W AM2>)))
End
```

Finally, the Univ_move specification is a copy of the "mov" instruction.

6.2.3 Error messages

The PAGODE constructor checks whether the Univ_assign, Univ_store, Univ_move exist in the target machine specification. When this condition does not hold one of the following error messages is emitted :

- Error : There is no Univ_load instruction, check your target machine specification.
- Error : There is no Univ_store instruction, check your target machine specification.
- Error : There is no Univ_move instruction, check your target machine specification.

6.3 Input and output of the prepare engine

6.3.1 Input of the prepare engine

```
seq (
```

```
    Univ_assign ( disp_am <gpr %fp> <value_13 -36>
    ,       temporary_am <temporary tmp8> )
    Univ_assign ( value_13_am <value_13 12>
    ,       temporary_am <temporary tmp9>)
    Univ_assign ( disp_am <gpr %fp> <value_13 -32>
    ,       temporary_am <temporary tmp10> )
    fmuld ( temporary_am <temporary tmp9>
    ,       temporary_am <temporary tmp10>
    ,       temporary_am <temporary tmp11>)
    add ( temporary_am <temporary tmp8>
    ,       temporary_am <temporary tmp11>
    ,       temporary_am <temporary tmp12>)
    Univ_assign ( temporary_am <temporary tmp12>
    ,       disp_am <gpr %fp> <value_13 -28>)
    ba ( relative_am <code_label l2>)
)
```

6.3.2 Output of the prepare engine

All Univ_assign operators have been overloaded by operators on which there exists attributes for scheduling, i.e Univ_load, Univ_store and Univ_move.

```
seq (
    Univ_load ( disp_am <gpr %fp> <value_13 -36>
    ,       temporary_am <temporary tmp8> )
    Univ_move ( value_13_am <value_13 12>
    ,       temporary_am <temporary tmp9>)
    Univ_load ( disp_am <gpr %fp> <value_13 -32>
    ,       temporary_am <temporary tmp10> )
    fmuld ( temporary_am <temporary tmp9>
    ,       temporary_am <temporary tmp10>
    ,       temporary_am <temporary tmp11>)
    add ( temporary_am <temporary tmp8>
    ,       temporary_am <temporary tmp11>
    ,       temporary_am <temporary tmp12>)
    Univ_store ( temporary_am <temporary tmp12>
    ,       disp_am <gpr %fp> <value_13 -28>)
    ba ( relative_am <code_label l2>)
)
```

Chapter 7

Creating a code generator

7.1 Installation of the PAGODE system

Since the PAGODE system strongly relies on the FNC2 and SYNTAX systems, the installation of these two systems must be done first and in your environment setting the variables **$sx** and **$f2** . Then, select a directory in your file system hierarchy for the installation of the PAGODE system. Add to your environment, the **$PG** variable set to the value of the path to this directory. Add to your PATH, two paths:

- **$PG**
- **$PG/scala**

7.2 Creating a code generator

In the sequel, the two following notations will be used:

- *target_option* allows to specify the category of target machine: it can take either the value *risc* or the value *cisc*.
- the *target_machine* is the name of the target machine, this name must be the prefix of the file that describes the SCALA specification.

1. Create a directory where the code generator will be located for instance 'code_generator'.
2. Add to your environment, the **$BCG** variable that defines the access path to this directory.

   Then add to your PATH the following path **$BCG/commands** that allows to call various engines of the code generator.
3. Go to the directory 'code_generator' denoted by **$BCG**.
4. Type:

   install_code_generator *target_option*

   If the *target_option* is omitted or invalid, you will be prompted thus:

The command is given up - Give the the target option (risc or cisc).

then you will then be prompted :

The code generator will be created in the current working directory :
Do you really want to install your code generator here (y/n)?

The following hierarchy is built :

spec/

commands/

instruction_selection/ :

- bin/
- incl/
- lib/
- src/

binding/ :

- bin/
- f2aux/
- incl/
- incl_fnc2/
- lib/
- spec/
- src/
- src_fnc2/
- tmp/

reg_alloc/ :

- incl/
- lib/
- src/
- tables/

assign_rewrite/ :

- bin/
- incl/
- lib/
- src/

emit/ :

- bin/
- f2aux/
- incl/
- incl_fnc2/
- lib/
- spec/
- src/
- src_fnc2/
- tmp/

If the value of *target_option* is **risc**, the preceding hierarchy includes :

prepare/ :

- bin/
- f2aux/
- incl/
- incl_fnc2/
- lib/
- spec/
- src/
- src_fnc2/
- tmp/

scheduling/ :

- bin/
- f2aux/
- incl/
- incl_fnc2/
- lib/
- spec/
- src/
- src_fnc2/
- tmp/

5. in the directory spec/ put the two following specifications :

   - the target machine specification written in PAGODE. It must be included in a file called *target_machine*.**scala**.

- the specification of the tokens of the LIR described by regular expressions. It must be included in a file called *target_machine*.**lecl**.

6. In order to build the tables, go to the level of the code_generator, i.e into the $BCG directory and type the command:

   constructor *target_option target_machine*

   If the *target_option* is omitted or invalid, you will be prompted thus:

   The command is given up - Give the target option (risc or cisc).

   If the *target_machine* is omitted, you will be prompted:

   The command is given up - Give the target machine as second parameter.

7. In order to generate automatically the whole back-end engines corresponding to the previous choice of the target option, type the command:

   make_back_end

   If you have rather to generate the back end modules step by step, execute the following points:

8. In order to generate the binding module, go to the directory binding/ and type the command:

   makebinder

9. In order to build the instruction selector, go to the directory instruction_selection/ and type the command:

   makeinstruction_selector

10. Only for the risc option, in order to prepare the integration between the register allocation and the scheduling, go to the directory prepare/ and type the command:

    makepreparer

11. In order to build the register allocator, go to the directory reg_alloc/ and type the command:

    makeregister_allocator

12. In order to build the universal assign rewritor, go to the directory assign_rewrite/ and type the command:

makeassign_rewritor

13. Only for the risc option, in order to build the scheduler, go to the directory scheduling/ and type the command :

    makescheduler

14. In order to build the emit engine, go to the directory emit/ and type the command :

    makeemitter

7.3 Code generation

1. Create a directory where the code generation will be performed for instance "generation".
2. Go to this directory.
3. Type the command :

   install_code_generation *target_option*

   If the *target_option* is omitted or invalid, you will be prompted thus :

   The command is given up - Give the target option (risc or cisc).

   Then you will then be prompted :

   Code generation will take place in the current working directory:
   Do you really want to install your code generator here (y/n)?

   The following hierarchy is built :

   - terms/ – PMIR terms input of the instruction selection process
   - selection_output/ – output of the instruction selection step and for the cisc option : input to binder, for the risc option : input of the prepare step.
   - prepare_output/ – output of the prepare step and input to the first pass of scheduling (for the risc option only).
   - scheduling_output1/ – output of the first pass of scheduling and input to the binding engine.
   - regalloc_input/ – output of the binding step and input of the register allocation step.
   - regalloc_output/ – output of the register allocation step and input of the assign rewritor step.

- convert_output/ – output of the assign rewritor step. When dealing with a risc machine it is the input of the second pass of scheduling, otherwise it is the input of the emit engine.
- scheduler_output2/ – output of the second pass of scheduling and input of the emit engine.
- emit/ – location of the assembly code.

4. To activate the various engines of the code generator, you must be in the directory created for code generation, i.e 'generation'.

   - Instruction selection :
     Type the command :

     instruction_selection *term*

     The result *term*.so is located into the subdirectory selection_output/.
   - Preparing the integration of register allocation and scheduling only for the risc option :
     Type the command :

     prepare *term*

     The result *term*.po is located into the subdirectory prepare_output/.
   - To perform the first pass of scheduling, type the command :

     scheduling_pass1 *term*

     The result *term*.so1 is located into the subdirectory scheduling_output1/.
   - Binding :
     - Call the binding step and usedef analysis typing the command :

       binding *term*
     - The results are in the subdirectory regalloc_input/ :
       term.ri : ouput term
       term.bind_no_alias : binding constraints table
       term.use_def_no_alias : use-def table
   - Register allocation :
     - Type the command :

       register_allocation *term*
     - You can specify some options : *-n* and *-a -n, -nh or -noheader.*
       When developping a back end, it may appear comfortable to test a sequence of instructions that are a "procedure body" without bothering about the instructions that make the prologue of the procedure. The option -n stands for no header for procedure. The default is with header and has not to be specified.
       -a or -area: The strategy of spill code is the other option; the option "spill in area" must be specified by the option -a . The default option performs the "spill in stack" strategy.
     - The result *term*.ro is located into the subdirectory regalloc_output/.

- in order to rewrite a universal assign into an actual instruction and establish necessary conversion,
  - Type the command:

    assign_rewrite *term*
  - The result *term*.co is located into the subdirectory convert_output/.
- To perform the second pass of scheduling, type the command:

  scheduling_pass2 *term*

  The result *term*.so2 is located into the subdirectory scheduling_output2/.
- To emit the assembly code, type the command:

  emit *term*

  The result *term*.s is located into the subdirectory emit/.

5. To activate the various engines of the code generator in one step, go to the directory created for code generation and type:

   cg *term*

   If the *term* is omitted, you will be prompted thus:

   The command is given up - Give the PMIR term name as parameter.

   Each intermediate result is also located into its dedicated subdirectory. The options of the register_allocation command are also valid for the cg command.

Bibliography

[BJ 87] Boullier P., Deschamp Ph.: A new Error Repair Recovery Scheme for Lexical and Syntactic Analysis. *Science of Computer Programming* December 1987, **9**, 4 pp 271-286.

[CH 90] Chow F., Hennessy J.L.: The priority-Based Coloring Approach to Register Allocation. *Transactions on Programming Languages and Systems* Vol. 12, 4, October 1990, pp 501-536.

[DMR 90a] Despland A., Mazaud M., Rakotozafy R.: PAGODE: A back end generator using attribute abstract syntaxes and term rewritings. *Proceedings of Compiler Compilers*, Schwerin, Germany, October 1990, ed in LNCS no 477 pp 86-105.

[DMR 90b] Despland A., Mazaud M., Rakotozafy R.: Using rewriting techniques to produce code generators and proving them correct. *Science of Computer Programming* December 1990, **15**. 54 pp 15-54.

[GM 86] Gibbons P. B., Muchnick S. S.: Efficient Instruction Scheduling for a Pipelined Architecture. *Proceedings of the SIGPLAN'86, pp 11-16.*

[HO 82] Hoffman C.M., O'Donnell M.J.: Pattern matching in trees. *Journal of ACM* Vol. 29 ,1 . January 1982, pp 68-95.

[RR 92] Ramesh R., Ramakrishnan I.V.: Nonlinear pattern matching in trees. *Journal of ACM* Vol. 39. 2, April 1992, pp 295-316.

Appendix A

Derivation chains for a SPARC PMIR

The following SPARC PMIR:

```
assign_W (
    const_value_13 ( denotation ( value_13. 0 ))
    , designates_address_W (index (
            cont_of_gpr_W ( denotation ( gpr, %fp ) )
            , const_value_13 ( denotation ( value_13. -4 )))))
```

produces the following derivation chain:

```
seq (
    mov (
          value_13_am_L <value_13 0>
          , temporary_am <temporary tmp0> )
    Univ_assign (
          temporary_am <temporary tmp0>
          , disp_am_W <gpr_W %fp> <value_13 -4> )
    )
```

provided that the PMIR is matched with the template of the move synthetic instruction specified as follows:

```
[--------------------------------------]
[  MOV mov reg_or_imm , rd             ]
[  T: GPR(rd) < - GPR(reg_or_imm)      ]
[--------------------------------------]
Instruction
            mov (<Reg_or_imm_access AM1> . <Gpr_access_W AM2>)
    Attributes
                    :
            $time_cost = 4
    Template
            assign_W
                    (src (<Reg_or_imm_access AM1>)
                    ,dst (<Gpr_access_W AM2>))
    End
```

The pattern matching of the PMIR succeeds with the previous template instruction form since:

1. the left operand:

   const_value_13 (denotation (value_13, 0)) is matched with the following access mode in source position:

```
Access_mode
    Canonical_form
            value_13_am (<value_13 val>)
    Attributes
                    ⋮
            $space_cost = 0
            $time_cost = 2
    Template
            Src = const_value_13 (<value_13 val>)
End
```

   and since the value_13_am access mode belongs to the right access class Reg_or_imm_access.

```
Access_class
    <Reg_or_imm_access AM>
            = gpr_am_W (<gpr_W reg>)
            = int_freg_am_W (<int_freg_W reg>)
            = value_13_am (<value_13 val>)
            = lo_label_am (<data_label lab>)
End
```

2. the right operand:

```
designates_address_W (index (
    cont_of_gpr_W ( denotation ( gpr, %fp ) )
    , const_value_13 ( denotation ( value_13, -4 )))
```

   is matched with the disp_am_W access mode in destination position.

```
[- - - - - - - - - - - - - - - - - - - - - - - -]
[ frame pointer with displacement                ]
[- - - - - - - - - - - - - - - - - - - - - - - -]
Access_mode
    Canonical_form
            disp_am!size (<gpr_W reg>, <value_13 val>)
    Attributes
                    ⋮
            $space_cost = 0
            $time_cost = 1
    Template
            index (cont_of_gpr_W(<gpr reg>)
                    , const_value_13(<value_13 val>))
```

```
    Instances
            size in {B, H, W, D}
End
```

But the disp_am_W access mode does not belong to the Gpr_access_W class

```
Access_class
    <Gpr_access!size AM>
            = gpr_am!size (<gpr!size reg>)
            = int_freg_am!size (<int_freg!size reg>)
    Instances
            size in {W, D}
End
```

that must be recognized to match the PMIR with the mov template.

The instruction selector looks for the interface specification and finds that the Gpr_access class with its related access mode gpr_am!size is specified in this section

```
[- - - - - - - - - - - - - - - - - - - - -]
[ Interface for the SPARC                 ]
[- - - - - - - - - - - - - - - - - - - - -]
Interface_declaration
    $Bind = {gpr, int_freg, float_freg}
    Access_class
            <Temporary_access AM>
                    = gpr_am!size(<gpr!size temp>)
                    = int_freg_am!size (<int_freg!size reg>)
                    = float_freg_am!size (<float_freg!size reg>)
            Instances
                    size in {W, D}
    End
    Univ_assign_equivalence
            Univ_assign : assign!size where size in {B, H, W, D}
                            $space_cost = 0, $time_cost= 1
End
```

It means that temporary_am plays the role of gpr_am that belongs to the Gpr_access class. Thus the instruction selector builds the first instruction of the derivation chain :

```
mov (
    value_13_am_L <value_13 0>
    , temporary_am <temporary tmp0> )
```

Since the initial destination of the PMIR term is not preserved by the previous instruction, the instruction selector produces a universal store of the contents of temporary tmp0 into this destination.

```
Univ_assign (
    temporary_am <temporary tmp0>
    , disp_am_W <gpr_W %fp> <value_13 -4> )
)
```

Appendix B

Spill code strategies

In order to illustrate the production of spill code on a rather small term, the number of data registers of the MC68000 has been reduced to two registers d0 and d1.

In this input LIR term, tmp9 is alive in block 3 and 5 and tmp12 is alive in block 4 and 5.

In order to show the effect of the size information on the production of spill code, the size information for tmp13 has been artificially modified to :

```
tmp13 =  [ [34,L]  ,[35,L]  ,[48,W] ].
```

```
2   link (
        areg_am_L <aregister_L a6>
        , immediate_dlab_am_W <data_label_W -L3>)
...
block 3
17  lab ( relative_clab_am_W <code_label_W l1:> )
18  Univ_assign (
        disp_am_L <aregister_L a6> <immediate_value_W 8>
        , temporary_am <temporary tmp6>)
19  addi_L (
        immediate_val_am_L <immediate_value_L 1>
        , temporary_am <temporary tmp6>)
20  Univ_assign (
        temporary_am <temporary tmp6>
        , temporary_am <temporary tmp7>)
21  asl_L (
        quick_am <quick_value 2>
        , temporary_am <temporary tmp7>)
22  Univ_assign (
        temporary_am <temporary tmp7>
        , temporary_am <temporary tmp8>)
23  sub_L (
        immediate_val_am_L <immediate_value_L 4>
        , temporary_am <temporary tmp8>)
24  Univ_assign (
        immediate_val_am_L <immediate_value_L -28>
        , temporary_am <temporary tmp9>)
25  add_L (
        temporary_am <temporary tmp8>
```

```
        , temporary_am <temporary tmp9>)
26  bne (
        relative_clab_am_W <code_label_W l2>)
block4
27  Univ_assign (
        disp_am_L <aregister_L a6> <immediate_value_W -8>
        , temporary_am <temporary tmp10>)
28  asl_L (
        quick_am <quick_value 2>
        , temporary_am <temporary tmp10>)
29  Univ_assign (
        temporary_am <temporary tmp10>
        , temporary_am <temporary tmp11>)
30  sub_L (
        immediate_val_am_L <immediate_value_L 4>
        , temporary_am <temporary tmp11>)
31  Univ_assign (
        immediate_val_am_L <immediate_value_L -52>
        , temporary_am <temporary tmp12>)
32  add_L (
        temporary_am <temporary tmp11>
        , temporary_am <temporary tmp12>)
33  bne (
        relative_clab_am_W <code_label_W l2>)
block 5
34  Univ_assign (
        dindex_am_when_L_L <aregister_L a6>
                <dregister_L tmp9> <immediate_value_B 0>
        , temporary_am <temporary tmp13>)
35  sub_L (
        dindex_am_when_L_L <aregister_L a6>
                <dregister_L tmp12> <immediate_value_B 0>
        , temporary_am <temporary tmp13>)
36  Univ_assign (
        disp_am_L <aregister_L a6> <immediate_value_W -4>
        , temporary_am <temporary tmp14>)
37  sub_L (
        immediate_val_am_L <immediate_value_L 2>
        , temporary_am <temporary tmp14>)
38  Univ_assign (
        temporary_am <temporary tmp14>
        , temporary_am <temporary tmp15>)
39  asl_L (
        quick_am <quick_value 2>
        , temporary_am <temporary tmp15>)
40  Univ_assign (
        temporary_am <temporary tmp15>
        , temporary_am <temporary tmp16>)
41  sub_L (
        immediate_val_am_L <immediate_value_L 4>
        , temporary_am <temporary tmp16>)
42  Univ_assign (
        immediate_val_am_L <immediate_value_L -80>
```

```
        , temporary_am <temporary tmp17>)
43  add_L (
        temporary_am <temporary tmp16>
        , temporary_am <temporary tmp17>)
44  Univ_assign (
        dindex_am_when_L_L <aregister_L a6>
            <dregister_L tmp17>
            <immediate_value_B 0>
        , temporary_am <temporary tmp18>)
...
48  Univ_assign (
        temporary_am <temporary tmp13>
        , temporary_am <temporary tmp21>)
...
54  equ (
        immediate_dlab_am_W <data_label_W L3>
        , immediate_val_am_W <immediate_value_W 80>)
```

The temporaries are colored to the following registers:

live-range	Split	reg name	reg class
tmp10	0	d0	dregister_L
tmp11	0	d0	dregister_L
tmp12	0	d0	dregister_L
tmp12	2	d1	dregister_L
tmp13	-	-	dregister_L
tmp14	0	d0	dregister_L
tmp15	0	d0	dregister_L
tmp16	0	d1	dregister_L
tmp17	0	d0	dregister_L
tmp9	0	d0	dregister_L
tmp9	1	d1	dregister_L

Since tmp9 is alive in block 3 and 5 and tmp12 is alive in block 4 and 5, they can be colored to different registers in different blocks. Notice that the live range of tmp9 has been split into two "sub" live ranges and tmp9 is colored respectively to d0 and d1.

Therefore tmp9 has to be spilled at the end of block 3 and loaded at the beginning of block 5.

tmp12 has to be spilled at the end of block 4 and loaded at the beginning of block 5.

No more register is available for tmp13, it has to be spilled. Since tmp13 is defined in instruction 35 and used in instruction 48, it has to be stored after 35 and loaded before 48.

Thus, tmp9,tmp12 and tmp13 must be assigned to memory locations. On this small example, no other temporaries need such memory locations because tmp13 is defined when register d1 is not yet used and used at the end of the block when register d1 is no more used.

This term has been run with the two strategies. When choosing the spill in stack strategy, the following output is produced:

Before spilling, the size of the stack was :

```
54  equ (
        immediate_dlab_am_W <data_label_W L3>
        , immediate_val_am_W <immediate_value_W 80>)
```

After spilling, the size is increased in order to hold 3 temporaries the size of which is a long word (tmp9, tmp12, tmp13). They get respectively offset -84,-88,-92.

```
54  equ (
        immediate_dlab_am_W <data_label_W L3>
        , immediate_val_am_W <immediate_value_W 92>)
```

Instruction 25 is the last time tmp9 is defined in block 3. Thus after this instruction, the spill instruction is produced according to one of the instructions declared by the **store** keyword:

```
    move_L (
        dreg_am_L<dregister_W d1> ,
        disp_am_L<aregister_L a6><immediate_value_W -84>)
```

The first use of tmp9 in block 5 is in instruction 34. It has to be loaded into a register before this instruction.

```
34  move_L (
        disp_am_L<aregister_L a6><immediate_value_W -84> ,
        dreg_am_L<dregister_L d0>)
    Univ_assign (
        dindex_am_when_L_L<aregister_L a6>
            <dregister_L d0><immediate_value_B 0> ,
        dreg_am_L<dregister_L d0>)
```

When choosing the spill in stack strategy, the following output is got:

```
1   lab (
        relative_clab_am_W<code_label_W terme5_compute:>)
2   link (
        areg_am_L<aregister_L a6> ,
        immediate_dlab_am_W<data_label_W -L3>)
...
17  lab (
        relative_clab_am_W<code_label_W l1:>)
18  Univ_assign (
        disp_am_L<aregister_L a6><immediate_value_W 8> ,
        dreg_am_L<dregister_L d0>)
19  addi_L (
        immediate_val_am_L<immediate_value_L 1> ,
        dreg_am_L<dregister_L d0>)
20
21  asl_L (
        quick_am<quick_value 2> ,
        dreg_am_L<dregister_L d0>)
```

```
22
23  sub_L (
        immediate_val_am_L<immediate_value_L 4> ,
        dreg_am_L<dregister_L d0>)
24  Univ_assign (
        immediate_val_am_L<immediate_value_L -28> ,
        dreg_am_W<dregister_W d1>)
25  add_L (
        dreg_am_L<dregister_L d0> ,
        dreg_am_W<dregister_W d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a6><immediate_value_W -84>)
26  bne (
        relative_clab_am_W<code_label_W l2:>)
27  Univ_assign (
        disp_am_L<aregister_L a6><immediate_value_W -8> ,
        dreg_am_L<dregister_L d0>)
28  asl_L (
        quick_am<quick_value 2> ,
        dreg_am_L<dregister_L d0>)
29
30  sub_L (
        immediate_val_am_L<immediate_value_L 4> ,
        dreg_am_L<dregister_L d0>)
31  Univ_assign (
        immediate_val_am_L<immediate_value_L -52> ,
        dreg_am_W<dregister_W d1>)
32  add_L (
        dreg_am_L<dregister_L d0> ,
        dreg_am_L<dregister_L d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a6><immediate_value_W -88>)
33  bne (
        relative_clab_am_W<code_label_W l2:>)
34  move_L (
        disp_am_L<aregister_L a6><immediate_value_W -84> ,
        dreg_am_L<dregister_L d0>)
    Univ_assign (
        dindex_am_when_L_L<aregister_L a6>
            <dregister_L d0><immediate_value_B 0> ,
        dreg_am_L<dregister_L d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a6><immediate_value_W -92>)
35  move_L (
        disp_am_L<aregister_L a6><immediate_value_W -88> ,
        dreg_am_L<dregister_L d0>)
    move_L (
        disp_am_L<aregister_L a6><immediate_value_W -92> ,
        dreg_am_L<dregister_L d1>)
    sub_L (
```

```
        dindex_am_when_L_L<aregister_L a6><dregister_L d0><immediate_value_B 0> ,
        dreg_am_L<dregister_L d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a6><immediate_value_W -92>)
36  Univ_assign (
        disp_am_L<aregister_L a6><immediate_value_W -4> ,
        dreg_am_L<dregister_L d0>)
37  sub_L (
        immediate_val_am_L<immediate_value_L 2> ,
        dreg_am_L<dregister_L d0>)
38
39  asl_L (
        quick_am<quick_value 2> ,
        dreg_am_L<dregister_L d0>)
40  Univ_assign (
        dreg_am_L<dregister_L d0>)
        dreg_am_L<dregister_L d1>)
41  sub_L (
        immediate_val_am_L<immediate_value_L 4> ,
        dreg_am_L<dregister_L d1>)
42  Univ_assign (
        immediate_val_am_L<immediate_value_L -80> ,
        dreg_am_L<dregister_L d0>)
43  add_L (
        dreg_am_L<dregister_L d1> ,
        dreg_am_L<dregister_L d0>)
44  Univ_assign (
        dindex_am_when_L_L<aregister_L a6><dregister_L d0><immediate_value_B 0> ,
        areg_am_L<aregister_L a2>)
...
48  move_W (
        disp_am_W<aregister_L a6><immediate_value_W -92> ,
        dreg_am_W<dregister_W d1>)
    Univ_assign (
        dreg_am_W<dregister_W d1>)
        dreg_am_L<dregister_L d0>)
...
54  equ (
        immediate_dlab_am_W<data_label_W L3> ,
        immediate_val_am_W<immediate_value_W 92>)
```

The non numbered instructions are the instructions added by the spill code process. When choosing the spill in area strategy, the following output is got:

```
1   lab (
        relative_clab_am_W<code_label_W terme5_compute:>)
2   movea_L (
        immediate_dlab_am_W<data_label_W area0>),
        areg_am_L<aregister_L a6>
    link (
        areg_am_L<aregister_L a6> ,
```

```
        immediate_dlab_am_W<data_label_W -L3>)
...
17  lab (
        relative_clab_am_W<code_label_W l1:>)
18  Univ_assign (
        disp_am_L<aregister_L a6><immediate_value_W 8> ,
        dreg_am_L<dregister_L d0>)
19  addi_L (
        immediate_val_am_L<immediate_value_L 1> ,
        dreg_am_L<dregister_L d0>)
20
21  asl_L (
        quick_am<quick_value 2> ,
        dreg_am_L<dregister_L d0>)
22
23  sub_L (
        immediate_val_am_L<immediate_value_L 4> ,
        dreg_am_L<dregister_L d0>)
24  Univ_assign (
        immediate_val_am_L<immediate_value_L -28> ,
        dreg_am_W<dregister_W d1>)
25  add_L (
        dreg_am_L<dregister_L d0> ,
        dreg_am_W<dregister_W d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a5><immediate_value_W 4>)
26  bne (
        relative_clab_am_W<code_label_W l2:>)
27  Univ_assign (
        disp_am_L<aregister_L a6><immediate_value_W -8> ,
        dreg_am_L<dregister_L d0>)
28  asl_L (
        quick_am<quick_value 2> ,
        dreg_am_L<dregister_L d0>)
29
30  sub_L (
        immediate_val_am_L<immediate_value_L 4> ,
        dreg_am_L<dregister_L d0>)
31  Univ_assign (
        immediate_val_am_L<immediate_value_L -52> ,
        dreg_am_W<dregister_W d1>)
32  add_L (
        dreg_am_L<dregister_L d0> ,
        dreg_am_L<dregister_L d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a5><immediate_value_W 8>)
33  bne (
        relative_clab_am_W<code_label_W l2:>)
34  move_L (
        disp_am_L<aregister_L a5><immediate_value_W 4> ,
        dreg_am_L<dregister_L d0>)
```

```
    Univ_assign (
        dindex_am_when_L_L<aregister_L a6>
            <dregister_L d0><immediate_value_B 0> ,
        dreg_am_L<dregister_L d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a5><immediate_value_W 12>)
35  move_L (
        disp_am_L<aregister_L a5><immediate_value_W 8> ,
        dreg_am_L<dregister_L d0>)
    move_L (
        disp_am_L<aregister_L a5><immediate_value_W 12> ,
        dreg_am_L<dregister_L d1>)
    sub_L (
        dindex_am_when_L_L<aregister_L a6><dregister_L d0><immediate_value_B 0> ,
        dreg_am_L<dregister_L d1>)
    move_L (
        dreg_am_L<dregister_L d1> ,
        disp_am_L<aregister_L a5><immediate_value_W 12>)
36  Univ_assign (
        disp_am_L<aregister_L a6><immediate_value_W -4> ,
        dreg_am_L<dregister_L d0>)
37  sub_L (
        immediate_val_am_L<immediate_value_L 2> ,
        dreg_am_L<dregister_L d0>)
38
39  asl_L (
        quick_am<quick_value 2> ,
        dreg_am_L<dregister_L d0>)
40  Univ_assign (
        dreg_am_L<dregister_L d0>)
        dreg_am_L<dregister_L d1>)
41  sub_L (
        immediate_val_am_L<immediate_value_L 4> ,
        dreg_am_L<dregister_L d1>)
42  Univ_assign (
        immediate_val_am_L<immediate_value_L -80> ,
        dreg_am_L<dregister_L d0>)
43  add_L (
        dreg_am_L<dregister_L d1> ,
        dreg_am_L<dregister_L d0>)
44  Univ_assign (
        dindex_am_when_L_L<aregister_L a6><dregister_L d0><immediate_value_B 0> ,
        areg_am_L<aregister_L a2>)
...
48  move_W (
        disp_am_W<aregister_L a5><immediate_value_W 12> ,
        dreg_am_W<dregister_W d1>)
    Univ_assign (
        dreg_am_W<dregister_W d1>)
        dreg_am_L<dregister_L d0>)
...
54  equ (
```

```
    immediate_dlab_am_W<data_label_W L3> ,
    immediate_val_am_W<immediate_value_W 92>)
lab (
    relative_clab_am_W<code_label_W area0:>)
data_storage_B (
    immediate_val_am_W<immediate_value_W 12>)
```

Index

Access_class, 15
Attributes, 16
Branch_init_read_cycle, 40
Canonical_form, 12, 15, 18
Convert declaration, 36
Delay_table, 39
Denotation, 9
Directive, 18
Dst, 13
Instances, 9, 13, 15, 17
Interface declaration, 19
Interlock_table, 42
Label_class, 10
MEM, 8, 10
Operations, 9
Src, 13
Symbolic_notation, 7
Template, 12, 13, 15
UNIV_INTEGER, 11
UNIV_LABEL, 10
Union, 9
Univ_assign_equivalence, 20
Univ_assign, 19, 22
Univ_load, 49
Univ_move, 49
Univ_store, 49
Value_class, 11
$Base, 9
$Bind, 20
$Booking_table, 42
$IOP_cycle_shift, 43
$IOP_patched_ins, 43
$delay_slot, 45
$end_write_cycle, 40
$executed_slot, 45
$fmt, 12, 16
$format, 16
$free, 8
$init_read_cycle, 40
$length, 12, 16
$modification, 16, 22, 32
$side_effect_OK, 41
$side_effect_write_cycle, 41
$space_cost, 12
$time_cost, 12, 16
$use_delay_table, 40, 41
alias, 8
always, 45
binary_arith, 16
branch, 17
case, 18
cell_constructor is, 9
cond_branch, 17
convert, 17
def_stack_size, 35
dereference is, 9
directive, 17
double, 10
load, 16, 34
never, 45
noop, 17
proc_call, 17
put_label, 17
return_proc, 17
special_arith, 17
spill code, 34
spill_in_area, 34
spill_in_stack, 34
spill, 16
stack_link, 35
store, 16, 34
taken, 45
temporary, 22
test, 17
unary_arith, 16
untaken, 45
when, 14
SCALA , 3, 6

access class, 7, 15
access mode, 7, 11–13

binder, 28, 29, 48
binding, 3, 28, 29, 31, 49
booking table, 41

canonical form, 12, 15, 21
cell constructor, 9, 12

delay_table, 39, 40
dereference, 9, 12
derivation chain, 21, 22, 24
destination operand, 11
directive, 18

instruction, 7, 15
interlock table, 42

label class, 10
LIR, 3, 21, 31

PMIR, 3, 12, 21, 23, 32
prepare, 49

register allocation, 3, 8, 31, 48

scheduling, 3, 38, 48–50
source operand, 11
storage base, 6–8
storage class, 6, 9, 10
symbolic notation, 8

template, 3, 12, 14, 15, 17, 21
temporary, 3, 19, 20, 22, 28

value class, 11

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399